The Dissertation Committee for Muqeet Ali
certifies that this is the approved version of the following dissertation:

A Second Generation of Nonrepudiation Protocols

Committee:

Mohamed G. Gouda , Supervisor

Lili Qiu

Aloysius K. Mok

Ehab Elmallah

A Second Generation of Nonrepudiation Protocols

by

Muqeet Ali

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2017

This thesis is dedicated to my parents.

Acknowledgments

I am grateful to many people who have helped me during my graduate studies at UT Austin. I am most grateful to my adviser Prof. Mohamed Gouda, who has helped me immensely in my research, and guided me through thick and thin to accomplish my research goals. His attention to detail always helped me find better ways to write and present my research. I am very grateful to him for introducing me to the area of nonrepudiation protocols, which became the topic of my PhD research. He always showed enthusiasm about research, and helped me greatly in every step of the research process. He has also given me sound advice about non-academic matters, and remains committed to all his students.

Other than my superviser, there are many people who I wish to thank. I wish to thank Prof. Lili Qiu, Prof. Al Mok and Prof. Ehab Elmallah for agreeing to serve on my committee, and for providing useful feedback. I am thankful to Mike Walfish with whom I did some research during the beginning of my graduate studies. His energy and enthusiasm got me excited about research. I am also very thankful to my colleague Rezwana Reaz for her collaboration on the papers I wrote during graduate studies.

I must thank my undergraduate mentor, Prof. Zartash Uzmi, who inspired me to do research and pursue a PhD degree. His faith in me encouraged

me to apply to PhD programs in USA. Also, I must mention the encouragement and support that I have received from my family which helped me greatly during graduate studies. My brother, Muneeb Ali, who is himself a PhD student in USA always had very sound advice for me. He always took out time for me, and helped me in every way he could.

My parents and sister, Saira, have always shown great faith in me, and always encouraged me to do better. I would not have been able to complete my graduate studies without their support, love, and prayers. I would also like to thank my wife, Sara, for her patience and constant encouragement. Her support was always present for me, and helped me get through the difficult times.

I would also like to thank my friends at UT Austin who helped me enjoy my time here. I would like to thank Amber Hassaan, Rashid Kaleem, Owais Khan, Akbar Mehdi, Mubashir Adnan Qureshi, Ghufran, Talha and others, for their company.

In the end, I would like to thank Allah, whose guidance and generosity, made it possible for me to do research and complete my dissertation.

A Second Generation of Nonrepudiation Protocols

Publication No. _____

Muqeet Ali, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Mohamed G. Gouda

A nonrepudiation protocol from party S to party R performs two tasks. First, the protocol enables party S to send to party R some text x along with sufficient evidence (that can convince a judge) that x was indeed sent by S. Second, the protocol enables party R to receive text x from S and to send to S sufficient evidence (that can convince a judge) that x was indeed received by R. The first generation of nonrepudiation protocols were published in the period 1996-2000. In this dissertation, we design a second generation of nonrepudiation protocols that enjoy several interesting properties.

First, we identify in this dissertation a special class of nonrepudiation protocols, called two-phase protocols. The two parties, S and R, in each two-phase protocol execute the protocol as specified until one of the two parties receives its needed proof. Then and only then does this party refrain from sending any more message specified by the protocol because these messages

only help the other party complete its proof. We show that the execution of each two-phase protocol is deterministic and does not require synchronized real-time clocks. We also show that each two-phase protocol needs to involve a trusted third party T beside the two original parties, S and R.

Second, we show that if party R in a two-phase protocol has a real-time clock and knows an upper bound on the round trip delay from R to S and back to R, then the two-phase protocol does not need to involve a trusted third party T.

Third, we design a nonrepudiation protocol for transferring file F from a sender S to a receiver R over a cloud C. This protocol is designed such that there is no direct communication between parties S and R. Rather all communications between S and R are carried out through cloud C. In this protocol parties S and R do not need to store a local copy of file F and the proofs that are needed by the two parties S and R (the only copy of file F and the proofs is stored in cloud C).

Fourth, we design a new nonrepudiation protocol from S to R over C where some of the proofs stored in cloud C get lost. This new protocol has an interesting stabilization property which ensures that when some of the proofs get lost, and one party can get the needed proofs but the other party cannot get its needed proofs from cloud C, then eventually, neither party is able to receive its needed proofs from cloud C.

Fifth, we design a nonrepudiation protocol for transferring files from a

sender S to a subset of potential receivers $\{R.1, R.2, \ldots, R.n\}$ over a cloud C. The protocol guarantees that after each file F is transferred from sender S to a subset of the potential receivers, then (1) each receiver Ri in the subset ends up with a proof that file F was indeed sent by sender S to Ri, and (2) sender S ends up with a proof that file F was indeed received from S by each receiver Ri in the subset.

Table of Contents

Chapter 1

Introduction

A nonrepudiation protocol from party S to party R performs two tasks. First, the protocol enables party S to send to party R some text x along with sufficient evidence (that can convince a judge) that x was indeed sent by S. Second, the protocol enables party R to receive text x from S and to send to S sufficient evidence (that can convince a judge) that x was indeed received by R. Nonrepudiation protocols have important applications in the world of e-commerce and cybersecurity. Whenever, an item is exchanged over the internet, there are concerns regarding whether the sender or the receiver of the item can later repudiate having sent or received the item. Nonrepudiation protocols would allow the sender of an item, to obtain a proof that the receiver indeed received the item. Similarly, the protocol would allow the receiver of an item, to obtain a proof that the sender indeed sent the item. These proofs can be submitted to a judge which verifies the submitted proof and declares whether the proof is valid or not. The decision of the judge is legally binding.

Research in nonrepudiation protocols started in 1996 when Gollmann and Zhou published the first paper about nonrepudiation protocols [62]. This paper described the first nonrepudiation protocol which involved three parties:

party S, party R, and a trusted third party T. Next, we give a high-level description of the Gollmann-Zhou protocol.

The Gollmann-Zhou protocol consists of the following five steps:

- Party S sends an encrypted text C to party R, alongwith a proof that C was sent by S to R.
- Party R sends back an acknowledgement by sending a proof that C was received by R from S.
- Party S sends the key K, which can be used to decrypt C, to trusted third party T.
- Party R can retrieve the key K and proof that key was issued by T from party T.
- Party S can also retrieve the proof that key was issued by T from party T.

After executing five steps of the protocol, party S ends up with following two items: a proof that C was received by R from S from step 2, and proof that key was issued by party T from step 5. Both these items are required to prove to a judge that message M, whose encryption is C, was indeed received by party R from S.

Similarly, after five steps of the protocol, party R ends up with the following three items: a proof that C was sent by S to R from step 1, and

both key K and a proof that key was issued by T from step 4. These items are needed to prove to a judge that message M, was indeed sent by party S to R.

The proof that C was sent by S to R is signed using the private key of party S which only party S knows. Similarly, the proof that C was received by R from S is signed using the private key of party R which only party R knows. The proof that key K was issued by T is signed using the private key of party T which only party T knows, and this proof proves that key K was received by trusted third party T from party S.

After the publication of the Gollmann-Zhou protocol in 1996, there appeared many papers related to the topic of nonrepudiation. We refer to these papers which appeared in the period 1996-2000 as the first generation of nonrepudiation protocols. This generation of nonrepudiation protocols made many contributions to this area of research. Most notably, the idea of optimistic nonrepudiation protocol was introduced during this era [39, 40, 65]. In optimistic nonrepudiation protocols, the trusted third party is not involved in every execution of the protocol. Rather, the protocol is designed to work without the involvement of the trusted third party provided both parties, S and R, execute the protocol as specified. If a party, S or R, deviates from the specified behaviour, then the trusted third party is involved in the execution of the protocol, to ensure the correctness of the nonrepudiation protocol, i.e., either both parties S and R end up with the needed evidence, or neither of them ends up with its needed evidence. Hence, some executions of the nonre-

pudiation protocol involve the trusted third party T while other executions do not involve the third party T. This was a major improvement in the design of the nonrepudiation protocols, as the trusted third party T can easily become a bottleneck, and become a single point of failure. Optimistic nonrepudiation protocols allowed to minimize the involvement of the trusted third party T, and only involved party T when a party decided to deviate from its specified behavior. More details about optimistic nonrepudiation protocols is included in the related works chapter.

Another important contribution of the first generation of nonrepudiation protocols is the protocol presented by Markowitch and Yves Roggeman in 1999 [41]. This nonrepudiation protocol only involved two parties, S and R, without any involvement of a third party. While this protocol made an interesting contribution, it also made some strong assumptions about the computing power of the parties involved. The protocol made the assumption, that party S knows an upper bound on the computing power of party R. In particular, the assumption was that the minimum time party R takes to decrypt a message is greater than the upper bound on the round-trip delay from S to R, and back to S. This assumption was quite unrealistic since party R can always decrease the time it takes to decrypt a message by buying more computing power, e.g., by employing a supercomputer. This unrealistic assumption made the practical use of this protocol questionable. Also, this protocol made the readers think whether a protocol can be designed which does not involve a trusted third party, and also does not rely on unreasonable set of assumptions. An

important contribution of this thesis, which presents the second generation of nonrepudiation protocols, is the design of a nonrepudiation protocol from S to R that does not involve a trusted third party which does not rely on unreasonable set of assumptions.

1.1 Limitations of Previous Work

The first generation of nonrepudiation protocols suffered from some drawbacks and limitations. The first important drawback of this generation of nonrepudiation protocols is that there is no formal method of specifying nonrepudiation protocols. This lack of formalism in the specification of nonrepudiation protocols has made the task of verifying these protocols very hard. A number of nonrepudiation protocols have been shown to be incorrect after they were published [24, 25, 28]. Also, a lack of formal specification made these protocols unnecessarily complicated, for example, many protocols made use of a large number of different message types [39], which makes the protocol hard to reason about and prove correct. In Chapter 3, we address the question of whether a simple formal specification of nonrepudiation protocols is possible which allows for easy verification of nonrepudiation protocols. In Chapter 3, we introduce two-phase nonrepudiation protocols which are easy to specify and verify. We also design a number of two-phase protocols to show their wide applicability.

Another drawback of the first generation of nonrepudiation protocols is that protocols which were designed to involve only two parties, S and R,

without any involvement of a trusted third party, made unrealistic assumptions. The protocol by Markowitch and Roggeman, which we refer to as MR protocol, raised several questions about the design of nonrepudiation protocols that do not involve a trusted third party. The first question was whether it is possible to design a nonrepudiation protocol from S to R that does not impose the unrealistic assumption of party S knowing an upper bound on the computing power of party R? The second question was that is it possible to establish the necessary and sufficient conditions to design the nonrepudiation protocol from S to R that does not involve a trusted third party? Another question was whether the protocol can be extended from S to R to the case where the protocol is executed from S to $\{R.1, R.2, \cdots, R.n\}$ where each $R.i$ wants a proof that a text x was indeed sent by S to $R.i$, and S wants a proof that a text x was received by each $R.i$ from S. In this dissertation, we provide an answer to all these questions in Chapter 4. The protocol presented in Chapter 4 is the first nonrepudiation protocol from S to R that does not involve a trusted third party, and makes (only) a set of reasonable assumptions. Moreover, these assumptions are proved to be necessary and sufficient.

A third important limitation of the first generation of nonrepudiation protocols is that the protocols were designed assuming direct communication between S and R. Many emerging communication models do not necessarily allow a direct communication between the parties involved in the protocol. One such model involves the cloud where every communication is carried out through the cloud. In this thesis, we address the question of how to design

nonrepudiation protocols that do not allow direct communication between involved parties. In our model, every communication is carried out through the cloud. In Chapter 5, we discuss in detail the design of nonrepudiation protocol that transfers a file F from S to R in cloud systems. In this protocol, the only copy of file F and the needed proofs is stored in the cloud.

In Chapter 6, we design a new nonrepudiation protocol from S to R in cloud system where some of the proofs stored in cloud get lost. We first show that the protocol presented in Chapter 5 suffers from a compromised state when some of the proofs stored in the cloud get lost. In the compromised state of the protocol, one party is able to receive its needed proofs but the other party is not able to receive its needed proofs. We design a new nonrepudiation protocol which has an interesting stabilization property, which ensures that when some of the proofs stored in the cloud get lost, and one party is able to receive its needed proofs but the other party is not able to receive its needed proofs, then eventually, neither party S or R is able to receive its needed proofs. This is the first ever nonrepudiation protocol with stabilization property.

In Chapter 7, we extend the design of nonrepudiation protocol from S to R presented in Chapter 5 to work from sender S to a subset of potential receivers $\{R.1, R.2, \ldots, R.n\}$ over a cloud C. The protocol guarantees that after each file F is transferred from sender S to a subset of the potential receivers, then (1) each receiver Ri in the subset ends up with a proof that file F was indeed sent by sender S to Ri, and (2) sender S ends up with a proof that file F was indeed received from S by each receiver Ri in the subset.

1.2 Contributions

The drawbacks and limitations of the first generation of nonrepudiation protocols inspired the second generation of nonrepudiation protocols which are the subject of this dissertation. In the second generation of nonrepudiation protocols, the following key contributions are made:

- We have identified a class of nonrepudiation protocols known as two-phase nonrepudiation protocols. These protocols are deterministic, and do not make use of real-time clocks. Also these protocols only make use of two types of messages. These properties make the specification and verification of two-phase protocols easier. We outline several two-phase nonrepudiation protocols in Chapter 3, and also present their security analysis. Also, we prove that every correct two-phase nonrepudiation protocol needs to involve a trusted third party.

- We present the first provably correct nonrepudiation protocol from party S to party R that does not involve a trusted third party. We also present the necessary and sufficient conditions that need to be adopted to design such a protocol. The protocol is first presented in a model where no messages can get lost, and is later relaxed to handle the case where messages can get lost. Also, the protocol is extended from S to R to the case where it involves n+1 parties: S, $R.1$, $R.2$, $\cdots$, $R.n$. These contributions are presented in Chapter 4.

- We design nonrepudiation protocols in cloud systems where no direct communication is allowed between involved parties. In fact, every communication is carried out through the cloud. We present the details of the design of nonrepudiation protocol from S to R in this model, and discuss the advantages of these protocols. These details are included in Chapter 5.

- We present a nonrepudiation protocol from S to R in cloud sytems where some of the proofs stored in the cloud get lost. This protocol has interesting stabilization property which ensures that when some of the proofs stored in the cloud get lost, and one party is able to receive its needed proofs but the other party is not able to receive its needed proofs from the cloud then, eventually, neither party S or R is able to receive its needed proofs from the cloud. These details are presented in Chapter 6.

- We extend the design of nonrepudiation protocol from S to R presented in Chapter 5 to work from sender S to a subset of potential receivers $\{R.1,\ R.2,\ \ldots,\ R.n\}$ over a cloud C. These details are presented in Chapter 7.

It is noteworthy that in Chapter 3 we prove that every correct two-phase nonrepudiation protocol needs to involve a trusted third party, but later show in Chapter 4 that one can design a nonrepudiation protocol from S to R that does not involve a trusted third party. The reason for this difference is that in two-phase protocols no real-time clock is allowed in designing these protocols.

However, in Chapter 4 we relax this model and in fact party R is allowed to keep a real-time clock. This difference in model is very important to keep in mind while reading Chapter 3 and Chapter 4. We present the prior and related works in Chapter 2. In Chapter 8, we present our conclusions, and thoughts on future work.

Chapter 2

Prior Work

In this chapter, we discuss the previous work done in nonrepudiation protocols. Research in nonrepudiation protocols started in 1996 when Dieter Gollmann and Jianying Zhou published the first nonrepudiation protocol that involved three parties: party S, party R, and party T [31, 62, 64]. Party T is a trusted third party that is involved in every execution of the protocol. The role of this party is to make sure that both the parties receive their needed evidence. This protocol assumes that both parties S and R can communicate (eventually) with the trusted party T in order to receive their evidence. One major drawback of this protocol is that the trusted third party is involved in every execution of the protocol, and is therefore, prone to becoming a bottleneck and a single point of failure.

The nonrepudiation protocol from S to R was later generalized to work from S to multiple parties, $R.1$, $R.2$, $R.3$, $\cdots$, $R.n$ [29, 30, 44, 45]. In these protocols, party S wants a proof that party $R.i$ indeed received the text x sent from S to $R.i$. Also, party $R.i$ wants a proof that S indeed sent the text x to $R.i$. These protocols were designed using group encryption scheme which made sure that only those $R.i$ which acknowledged to have received the

encrypted text sent during first step of the protocol by party S are able to get the proof that party S indeed sent text x to party $R.i$. These protocols also involved the services of the trusted third party T. Therefore, these protocols also suffered from the same problems where the trusted third party becomes a bottleneck.

In order to overcome the drawback of having a trusted third party, several nonrepudiation protocols were proposed which intended to limit the involvement of the trusted third party [15, 26, 40, 65]. These protocols only involved the trusted third party when any party (S or R) refrained from sending a message as specified by the protocol. These protocols are known as optimistic nonrepudiation protocols. These protocols were designed "optimistically" assuming that most of the times, party S and party R, would follow the main protocol as specified. If both parties follow the main protocol as specified, then there is no need to involve the trusted third party. These protocols also included a recovery protocol which can be invoked by any of the two parties S and R. The recovery protocol is invoked by a party when it times-out and concludes that the other party has stopped executing the protocol as specified. The recovery protocol executed with the help of the trusted third party makes sure that both parties eventually terminate and either both receive the needed evidence, or neither receive their needed evidence. In these protocols it is assumed that the communication channel between S and trusted third party T is reliable. Similarly, it is assumed that the communication channel between R and T is reliable. Protocols were also proposed which extended the

ideas of optimistic nonrepudiation protocol from S to R to multiple parties which work from S to $R.1$, $R.2$, $\cdots$, $R.n$ [39]

Optimistic nonrepudiation protocols made an important contribution towards the design of nonrepudiation protocols by limiting the involvement of trusted third party. However, it did not answer the question whether nonrepudiation protocols can be designed which do not rely upon the services of a trusted third party in any execution of the protocol. In 1999, Markowitch and Roggeman published a nonrepudiation protocol (referred to as MR protocol) which only has two parties S and R with no involvement of a trusted third party T [41]. Although this protocol did not involve a trusted third party but it made some assumptions which can easily be violated in practice. The assumptions made by the MR protocol are as follows: 1) There is an upper bound ub on the round-trip delay from S to R, and back to S, 2) There is a lower bound lb on the time it takes party R to decrypt a message, and 3) $lb > ub$. In practice, party R can always decrease the value of lb by purchasing more computing power. Therefore, the correctness of the MR protocol remained questionable. In this dissertation, we have presented a nonrepudiation protocol from S to R that does not involve a trusted third party under a set of reasonable assumptions [3]. More details about this protocol are presented in Chapter 4.

One important problem with the first generation of nonrepudiation protocols is that many protocols were shown to be incorrect after they got published [24, 25, 28]. This is especially true for optimistic nonrepudiation

protocols whose execution is highly non-deterministic, and involves many sub-protocols. One contribution of this dissertation, is that we present a class of nonrepudiation protocols known as two-phase which are inherently deterministic, and do not involve real-time clocks [2]. These protocols are easier to specify and verify as compared to a protocol which is not two-phase. Two-phase nonrepudiation protocols are discussed in depth in Chapter 3.

Many authors have explored the area of formally verifying the correctness of nonrepudiation protocols [10, 32, 35, 37, 58]. The authors have used theorem proving techniques to study the security properties of nonrepudiation protocols in [10, 58]. In [35] authors use the PRISM model checker to analyse the properties of nonrepudiation protocol that does not involve a trusted party [41]. In [37] authors have formally specified nonrepudiation protocols using spi calculus. In [32] authors use the model checker MOCHA to verify several nonrepudiation protocols.

Nonrepudiation has some important applications. Perhaps the most well-known of these applications is certified email protocols [46, 47]. The first well-known certified email protocol was published in 1996 by Dieter Gollmann and Jianying Zhou [63]. This paper introduced the requirements and architecture of certified email, and also gave a protocol which involved n mail servers in addition to the sender S and recipient R of the email. The protocol ensured that the party S either gets a proof that the email was received by R, or it gets informed that the email could not be delivered. The drawback of this proposal was that the mail servers need to be trusted. Later, certified email

protocols were developed which limited the role of the trusted party.

In 1998, a new certified email protocol was published by Bruce and James which only required the involvement of a time stamped public forum (e.g., New York Times) instead of a fully trusted third party [54]. Other proposals were made to limit the involvement of the trusted third party such that not every execution of the certified email protocol involves the trusted third party [6, 21]. This area of research is similar to optimistic nonrepudiation protocols as discussed previously. In [21] the authors present a certified email protocol which is optimistic and works in a minimum number of rounds. In [6] the authors present a certified email protocol which uses agents as intermediaries which can conspire with any of the two parties S and R. The trusted third party is invoked only when the agents fail or misbehave.

Later many researchers generalized certified email protocols from two parties to multiple parties [20, 61, 67]. The first such protocol is presented in [20] which is an optimistic protocol and ensures that the sender S sends the same email to multiple recipients and also gets an acknowledgement from them. Later, Jianying Zhou and others [61, 67] presented an improved version of the multi-party certified email protocol with better security requirements.

Nonrepudiation has also found applications in a number of different areas such as cloud security [17, 19, 33, 59], smart grids [60], mobile billing [38, 48, 66] and pay TV systems [36, 55, 56]. The first paper about nonrepudiation in cloud systems involved (n+2) parties: party S, parties $R.1$, $R.2$, $\cdots$, $R.n$, and party C which is the cloud system [17]. This paper discussed about

how to provide nonrepudiated evidence to the recipient of the message that the message was indeed sent by the sender S through the cloud C. In this dissertation, we provide an improved nonrepudiation protocol in cloud systems which provide two clear advantages [1]. More details about this protocol are included in Chapter 7.

The work presented in [18] investigates the problem of file integrity in cloud systems, and gives a protocol to ensure that the file downloaded by party $R.i$ from a cloud system C, is indeed the same file F which party S uploaded to the cloud system. However, the authors in [18] do not present a protocol which satisfies the nonrepudiation requirements of the protocol presented in Chapter 7. Authors in [27, 51] presented a system that allows the customers of the cloud storage system, to detect and prove to a third party violations of four desirable security properties: confidentiality, integrity, write-serializability and read-freshness.

Nonrepudiation has also found applications in smart grid [9, 60]. In [60], authors have proposed a solution to the energy theft caused by altering meter readings. The authors porpose a protocol to ensure that smart meters at both the producer and consumer side remain within known error bounds.

Authors in [38, 48, 66] have addressed the problem of mobile billing. Disputes in mobile billing can arise when the provider of the service issues a bill which is in error, but the client has no evidence to prove that the bill has irregularities. In order to resolve such disputes, it is necessary that both provider and subscriber have evidence of the total call time made by the sub-

scriber on the given provider network. The problem gets more challenging as mobile users can also roam into foreign networks. Authors have proposed protocols to address these issues using digital signatures and hash chains. Mobile users need to submit a digital signature when requesting a call and release chained hash values during the session so that the call and its duration are undeniable.

Similar to mobile billing, there are applications of nonrepudiation to the area of pay TV systems [36, 55, 56]. In pay TV systems, a subscriber should only pay for the channels that he/she is registered for. In [36] authors propose a protocol that allows the subscriber to register the channels, and also suspend and/or change the channels the subscriber wishes to watch. Both the system administrator and subscriber get evidence of the channels registered.

Nonrepudiation has also found application in digital rights management (DRM) [43, 49]. Digital rights management ensure that copyright-protected content like music or books is only used by users who paid for it. Nonrepudiation is an important requirement for digital rights management. Without nonrepudiation parties can involve in malicious activities and there would be no evidence to prove such activities to a third party. Examples include, a service provider charging more money for a certain service, or a user stating that he/she did not buy a multimedia content. Protocols are proposed which can be integrated into existing DRM architectures, and allow for resolution of disputes between participants. Nonrepudiation has also found usage in vehicular networks and biometrics [34, 57].

The problem of designing nonrepudiation protocols is similar to the problem of designing contract signing protocols. In contract signing protocols each participant in the execution of the protocol is interested in getting a signature of all other parties on a known contract C. However, in contract signing protocols the contract C to be signed is known to all parties before the execution of contract signing protocol begins, while in nonrepudiation protocols the text x is only known to party S before the execution of the nonrepudiation protocol begins. Most contract signing protocols do make use of a trusted third party. See for example [4, 5, 50], which introduce the problem of contract signing, and present optimistic contract signing protocols.

Later, protocols were published to generalize from two-party to multi-party contract signing protocols [7, 8, 42, 68]. These protocols were also optimistic, and they also made improvements in the number of steps required to complete the execution of the protocol (see for example [7, 68]). Another interesting contribution to the area of contract signing protocols was the development of abuse-free contract signing protocols [22, 23]. In abuse-free contract signing protocols no party can be in a state where it can prove to a third party that it can choose to validate or invalidate the contract. Protocols which are not abuse-free, can result in one party (say Bob) proving to a third party that it is not yet committed to the contract but the other party (say Alice) is. Such protocols are not desirable as they give Bob the incentive to use the willingness of Alice to sign other contracts while never completing the contract signing with Alice.

The trusted third party in some of the published contract signing protocols is rather weak [11, 52]. For example, the trusted third party in Rabin's protocol [52] does not receive any messages from either of the two parties S or R. Rather, Rabin's third party periodically generates random numbers and sends them to the original parties S and R.

Some published contract signing protocols do not employ a trusted third party. See for example [12, 16]. However, the opportunism requirement that is fulfilled by these protocols is weaker than our opportunism requirement which is described in detail in Chapter 4. Our opportunism requirement requires that as soon as a party recognizes that it has received its evidence it terminates. In each of the protocols presented in [12, 16], a party (S or R) may continue to execute the contract signing protocol even after this party recognizes that it has collected all its needed evidence with the hope that the cost of processing its collected evidence can be reduced dramatically.

Chapter 3

Two-phase Nonrepudiation Protocols

The work that is presented in this chapter is based on the material published in [2] [1]. A nonrepudiation protocol from party S to party R performs two tasks. First, the protocol enables party S to send to party R some text x along with a proof (that can convince a judge) that x was indeed sent by S. Second, the protocol enables party R to receive text x from S and to send to S a proof (that can convince a judge) that x was indeed received by R. Only one of two outcomes is possible when the execution of a nonrepudiation protocol from S to R terminates. The first outcome is that both S and R obtain their required proofs. The second outcome is that neither S nor R obtains its required proof.

Most nonrepudiation protocols that have been published in the literature seem to suffer from the following three problems:

- Execution of the protocol is non-deterministic. This implies that execution of the protocol can follow anyone of several alternative paths. For example, execution of the nonrepudiation protocol in [26] can follow anyone of six alternative paths.

[1] Muqeet Ali, did ninety percent of the work in this paper

- Some parties involved in the protocol need to utilize synchronized real-time clocks. For example each of the three parties S, R, and T involved in the nonrepudiation protocol in [26] needs to utilize real-time clocks. Moreover, these three clocks need to be synchronized.

- Exchanged messages during execution of the protocol are of many distinct types. For example, there are nine distinct message types in the multiparty protocol in [39].

These three problems seem to complicate the specification and verification of nonrepudiation protocols. It is no accident that several nonrepudiation protocols have been shown to be incorrect after they have been published in the literature; see for example [24, 25, 28].

3.1 Our Contributions

In this chapter, we identify a rich class of nonrepudiation protocols, called two-phase protocols, that do not suffer from the above three problems:

- Execution of a two-phase nonrepudiation protocol is deterministic and so it is guaranteed to follow exactly one path.

- None of the parties involved in a two-phase protocol needs to utilize a real-time clock. Therefore, proving correctness of two-phase protocols does not require reasoning about real-time.

- Exchanged messages during execution of a two-phase protocol are of two types only.

Also in this chapter, we discuss how to specify and verify two-phase protocols and demonstrate that the task of specifying and verifying a two-phase protocol is simpler than the task of specifying and verifying a comparable protocol that is not two-phase.

3.2 Two-phase Protocols

The objective that party S achieves by participating in the execution of a nonrepudiation protocol from S to R is different from the objective that party R achieves by participating in the execution of the same protocol. The objective of party S is to obtain a proof that party R has received some text x that was sent earlier from S to R. The objective of party R is to obtain a proof that party S has sent some text x that was later received by R from S.

A nonrepudiation protocol from party S to party R is called two-phase iff execution of the protocol by parties S and R proceeds in two phases. In the first phase, the party (S or R) has not yet received its complete proof and so it continues to execute the protocol as specified. In the second phase, the party (S or R) has already received its complete proof and so it refrains from sending any more messages specified by the protocol because these messages only help the other party complete its proof. In other words, the two parties S and R in any two-phase protocol will always act in their own self-interests

during execution of the protocol.

Two-phase nonrepudiation protocols have the following advantages over nonrepudiation protocols that are not two-phase:

(a) Execution of a two-phase protocol is inherently deterministic, whereas execution of a protocol that is not two-phase is usually non-deterministic.

(b) The participating parties of a two-phase protocol do not need to have synchronized clocks, whereas parties of a protocol that is not two-phase need to have synchronized clocks.

(c) It follows from (a) and (b) above that specifying and verifying the correctness of a two-phase protocol are easier than specifying and verifying the correctness of a comparable protocol that is not two-phase.

One more advantage of two-phase protocols is that the exchanged messages in these protocols are of two types only: send-proof messages and receive-proof messages.

The rest of this chapter is organized as follows. In Section 3.3, we describe the (only) two types of messages that are exchanged during executions of two-phase protocols. Then in Section 3.4, we describe the syntax and semantics of a simple notation that can be used for specifying two-phase protocols. In Section 3.5, we present a procedure for verifying the correctness of two-phase protocols. In Section 3.6, we conclude that two-phase protocols that involve only two parties, namely S and R, are incorrect, and we present

a (correct) two-phase protocol that involves three parties: S, R and a trusted third party T.

In Section 3.7, we discuss the security analysis of our two-phase protocols. In particular, we show that each one of these protocols can defend against four types of attacks: malicious parties, message loss, collusion attacks, and replay attacks. Concluding remarks of this chapter are given in Section 3.8.

3.3 Send and Receive Proof Messages

The messages exchanged during the executions of a two-phase protocol are of two types: snd-proof messages and rcv-proof messages. In this section, we describe these two types of messages.

A snd-proof message has five fields: message type, text, from, to, and message signature. Consider for example the following snd-proof message:

(type: snd-proof,
text: x,
from: S,
to: R,
message signature: using the private key of S)

Because this message is signed using the private key of party S, only S could have constructed this message. In fact party S has constructed this message and sent it to party R in order to provide R with a proof that S is the one that

sent text x to R. (This should explain why we call this message a snd-proof message.)

At any point in the future, party R can show this snd-proof message to an (impartial) judge and can get this judge to declare that party S has indeed sent text x to party R and that party R has indeed received text x from party S.

Like a snd-proof message, a rcv-proof message has five fields: message type, text, from, to, and message signature. Consider for example the following rcv-proof message:

(type: rcv-proof,
text: x,
from: S,
to: R,
message signature: using the private key of R)

Because this message is signed using the private key of party R, only R could have constructed this message. In fact party R has constructed this message and sent it to party S in order to provide S with a proof that R is the one that received text x from S. (This should explain why we call this message a rcv-proof message.)

At any point in the future, party S can show this rcv-proof message, that it has received from party R, to an (impartial) judge and can get this

judge to declare that party S has indeed sent text x to party R and that party R has indeed received text x from party S.

From now on, each snd-proof message of the form: (type: snd-proof, text: x, from: S, to: R, message signature: using the private key of S) is written as:

snd-proof(x, S, R)

Also each rcv-proof message of the form: (type: rcv-proof, text: x, from: S, to: R, message signature: using the private key of R) is written as:

rcv-proof(x, S, R)

3.4 Specification of Two-phase Protocols

In this section, we describe a simple notation for specifying two-phase protocols that involve three parties S, R, and T. (The role of party T, which is called a trusted third party, in two-phase protocols is discussed in Section 3.6 below.)

In our notation, each two-phase protocol is specified as a finite and non-empty sequence of (atomic) *steps*, each of which is of the following form:

$P \rightarrow$ Q: msg.1, ..., msg.k

P denotes a party out of the three parties S, R, and T.
Q denotes a party, different from party P, out of the three parties S, R, and

T.

Each msg.i denotes either a snd-proof message or a rcv-proof message, as defined in the previous section.

Informally, execution of this step consists of party P sending to party Q the messages msg.1, ..., msg.k, and of party Q receiving these messages. (There are exceptions to this description; these are described below.)

Execution of a two-phase protocol consists of executing the steps of the protocol, one by one, starting from its first step. Execution of the protocol terminates when the last step in the protocol is executed or when any step in the protocol is executed and its execution causes execution of the protocol to terminate, as described below.

Associated with each two-phase protocol are three sets, called the S-proof set, the R-proof set, and the T-proof set. The S-proof set contains the snd-proof messages and rcv-proof messages that are received by party S during execution of the protocol. Similarly, the R-proof set (or the T-proof set, respectively) contains the snd-proof and rcv-proof messages that are received by party R (or party T, respectively) during execution of the protocol.

Before execution of a two-phase protocol starts, the three proof sets of the protocol are all empty. Then, as described below, executing each step adds some messages to at most one of these three proof sets.

The S-proof set of a two-phase protocol is said to be *complete* at some point during execution of the protocol iff the messages in the S-proof set at

this point are sufficient to prove (to a judge) that text x has been received by party R or by party T.

The R-proof set of a two-phase protocol is said to be *complete* at some point during execution of the protocol iff the messages in the R-proof set at this point are sufficient to prove (to a judge) that text x has been sent from S to R.

Note that the concept of the T-proof set of a two-phase protocol being complete is not defined.

Formally, the *execution* of a step

$P \rightarrow Q$: msg.1, ..., msg.k

in a two-phase protocol proceeds as follows

if P denotes party S
and-if the S-proof set of this protocol is complete before executing this step
then executing this step does not change any of the three proof sets but causes execution of the protocol to terminate

else-if P denotes party R
and-if the R-proof set of this protocol is complete before executing this step
then executing this step does not change any of the three proof sets but causes execution of the protocol to terminate

else executing this step adds the messages msg.1, ..., msg.k to the Q-

proof set of the protocol but does not change the other two proof sets

This formal definition of the execution of a step

$P \rightarrow Q$: msg.1, ..., msg.k

merits the following explanation.

Most of the time, execution of this step consists of party P sending the messages msg.1, ..., msg.k to party Q and of Q receiving these messages and adding them to the Q-proof set of the protocol. Later, party Q can use these messages in constructing its proof. There are two exceptions to this rule.

The first exception is when party P is in fact party S and the S-proof set of the protocol is already complete. In this case, party S recognizes that participating in the execution of this step does not help party S in constructing its proof (since its proof is already complete). In this case, execution of this step does not cause any of the messages msg.1, ..., msg.k to be sent or received, does not change any of the three proof sets of the protocol, but causes execution of the protocol to terminate.

The second exception is when party P is in fact party R and the R-proof set of the protocol is already complete. In this case, Party R recognizes that participating in the execution of this step does not help party R in constructing its proof (since its proof is already complete). In this case, execution of this step does not cause any of the messages msg.1, ..., msg.k to be sent or received, does not change any of the three proof sets of the protocol, but causes execution

of the protocol to terminate.

Note that there are two possible outcomes for the execution of a protocol step: either one or more messages are added to one of the three proof sets of the protocol, or the three proof sets remain unchanged and execution of the protocol terminates.

A two-phase protocol is *correct* iff the S-proof set and R-proof set of the protocol are both complete when execution of the protocol terminates.

3.5 Verification of Two-phase Protocols

The procedure for verifying the correctness of a two-phase protocol proceeds as follows:

1. Start with the three proof sets of the protocol (namely the S-proof set, the R-proof set, and the T-proof set) being empty.

2. Trace the three proof sets as the protocol steps are executed one by one starting with the first step in the protocol and until execution of the protocol terminates. (Recall that execution of the protocol terminates when the last step in the protocol is executed or when any protocol step, whose execution causes execution of the protocol to terminate, is executed.)

3. If both the S-proof set and R-proof set are complete when execution of the protocol terminates, then the protocol is correct. Otherwise, the protocol is incorrect.

In the remainder of this section, we present a two-phase protocol and use this verification procedure to show that this protocol is incorrect. Then, in the rest of the chapter, we present three other two-phase protocols and use this verification procedure to show that these protocols are all correct.

Two-phase Protocol 1:

Consider a two-phase protocol that consists of the following four steps.

$S \rightarrow R$: snd-proof(encrypt(x, key), S, R)

$S \leftarrow R$: rcv-proof(encrypt(x, key), S, R)

$S \rightarrow R$: snd-proof(key, S, R)

$S \leftarrow R$: rcv-proof(key, S, R)

The text in the message of the first step (and of the second step) consists of the "encryption" of text x using a "key" that only party S knows. The text in the message of the third step (and of the fourth step) consists only of the encryption "key".

The S-proof set of this protocol is complete when this set contains the two messages: rcv-proof(encrypt(x, key), S, R) and rcv-proof(key, S, R). Also the R-proof set of this protocol is complete when this set contains the two messages: snd-proof(encrypt(x, key), S, R) and snd-proof(key, S, R).

To check whether this protocol is correct, we trace the three proof sets of this protocol as the protocol steps are executed. This tracing leads to the following proof sets after the four protocol steps are executed:

S-proof = rcv-proof(encrypt(x, key), S, R)

R-proof = snd-proof(encrypt(x, key), S, R),

snd-proof(key, S, R)

T-proof = empty set

At this point, execution of the protocol terminates and S-proof is not complete but R-proof is complete. Because S-proof is not complete when execution of the protocol terminates, the protocol is not correct.

3.6 A Two-phase Protocol with a Third Party T

The two-phase protocol 1 is incorrect as follows. In the last step of this protocol, party R is supposed to send a message to party S. But the R-proof set of the protocol becomes complete before executing this last step and the message, that was supposed to be sent from R to S, is not sent. Thus, the S-proof set of the protocol remains incomplete when execution of the protocol terminates. This makes protocol 1 incorrect.

Theorem 3.1. *There is no correct two-phase nonrepudiation protocol from party S to party R that involves only parties S and R.*

Proof. Assume that we can design a correct two-phase nonrepudiation protocol that involves only the two parties S and R. Also, assume that the last step of this protocol is of the form:

$R \rightarrow S$: msg.1,. . ., msg.k

Now, there are two cases to consider:

Case 1: The R-proof set is not complete before executing this last step, then the R-proof set will not be complete when execution of this last step terminates, as execution of this last step will not change the R-proof set.
Case 2: The R-proof set is complete, then according to the specification of two-phase nonrepudiation protocol, the execution of this step does not change any proof set and causes execution of the protocol to terminate. We know that both the S-proof set and the R-proof set cannot be complete before the execution of this step. This is because if the S-proof set was complete before execution of this last step, then it was also complete before the execution of the previous step which was from S to R, and execution of that step would have resulted in termination of the protocol. It follows that S-proof set of the protocol was incomplete before execution of the last step, and will remain incomplete after execution of the last step.

In the above two cases, either the S-proof set or R-proof set is not complete when execution of the protocol terminates. This observation contradicts the assumption that this protocol was a correct two-phase nonrepudiation protocol.

A similar argument applies if the last step of the protocol is of the form:

$S \rightarrow R$: msg.1,. . ., msg.k

This proves that there is no correct two-phase nonrepudiation protocol from party S to party R that involves only parties S and R. □

In the remainder of this chapter, we only consider two-phase protocols

from S to R that involve three parties S, R and T. We assume that the third party T is not biased to either of the two parties S or R. This means that unlike the two parties S and R, which refrain from sending any message specified by the protocol when their respective proof sets are complete, party T never refrains from sending any message specified by the protocol. (See the definition of step execution in section 3.4 above.) Therefore, party T is often referred to as a trusted third party [62]. A trusted third party T is involved in each of the correct two-phase protocols discussed below.

Two-phase Protocol 2:

Consider a two-phase protocol that consists of the following three steps.

$S \rightarrow T$: snd-proof(x, S, R)

$S \leftarrow T$: rcv-proof(x, S, T), snd-proof(x, T, R)

$R \leftarrow T$: snd-proof(x, S, R)

In the first step, S sends a snd-proof(x, S, R) message to T which turns around and forwards the message to R in the third step. This message makes the R-proof set of the protocol complete. In the second step, T sends two messages to S. The first message acknowledges that T has received text x that was sent earlier by S and the second message promises that T will forward text x to R. These two messages acknowledge that text x that was sent earlier by S has been received or will shortly be received by R. These two messages make the S-proof set of the protocol complete.

It follows that the S-proof set of this protocol is complete when this set contains the two messages: rcv-proof(x, S, T) and snd-proof(x, T, R). The R-proof set of this protocol is complete when this set contains the message snd-proof(x, S, R).

To check that this protocol is correct, we trace the three proof sets of this protocol as the protocol steps are executed. The tracing leads to the following proof sets after the three protocol steps are executed:

S-proof = rcv-proof(x, S, T), snd-proof(x, T, R)

R-proof = snd-proof(x, S, R)

T-proof = snd-proof(x, S, R)

At this point, both S-proof and R-proof are complete. Because both S-proof and R-proof are complete when execution of the protocol terminates, the protocol is correct.

The two-phase protocol 2, though technically correct, has a problem. In this protocol, party S never gives party R a chance to express its agreement to participate in the protocol. Rather, party S starts by communicating directly with party T in order to complete the S-proof set (without soliciting any agreement from party R to participate in the protocol). Then party T turns around to communicate with party R in order to complete the R-proof set. In essence, party T "forces" party R to participate in the protocol.

To solve this problem, party S needs to start by communicating with party R (to ensure that R agrees to participate in the protocol) before S communicates with party T. This protocol is discussed next.

Two-phase Protocol 3:

Consider a two-phase protocol that consists of the following five steps.

$S \rightarrow R$: snd-proof(encrypt(x, key), S, R)

$S \leftarrow R$: rcv-proof(encrypt(x, key), S, R)

$S \rightarrow T$: rcv-proof(encrypt(x,key),S,R),
snd-proof(key, S, R)

$S \leftarrow T$: rcv-proof(key, S, T)

$R \leftarrow T$: snd-proof(key, S, R)

In the first step, party S sends a snd-proof(encrypt(x, key), S, R) message to party R to allow R to agree to participate in the protocol. In the second step, party R agrees to participate in the protocol by sending back to party S a rcv-proof(encrypt(x, key), S, R) message. This message acknowledges that R has received an encryption of text x using a key that only S knows. In the third step, S sends two messages to T. The first message proves that party R has agreed to participate in the protocol, and the second message carries the encryption key, which is to be forwarded from T to R in the fifth step. In the fourth step, the message rcv-proof(key, S, T) is sent from T to S in order to acknowledge that T has received the key that was sent by S in the third step.

It follows that the S-proof set of this protocol is complete when this set contains the two messages: rcv-proof(encrypt(x, key), S, R) and rcv-proof(key,

S, T). The R-proof set of this protocol is complete when this set contains the two messages: snd-proof(encrypt(x, key), S, R) and snd-proof(key, S, R).

To check that this protocol is correct, we trace the three proof sets of this protocol as the protocol steps are executed. This tracing leads to the following proof sets after the five protocol steps are executed:

$$\begin{aligned} S\text{-proof} &= \text{rcv-proof(encrypt}(x, \text{key}), S, R), \\ &\quad\ \text{rcv-proof(key}, S, T) \\ R\text{-proof} &= \text{snd-proof(encrypt}(x, \text{key}), S, R), \\ &\quad\ \text{snd-proof(key}, S, R) \\ T\text{-proof} &= \text{rcv-proof(encrypt}(x, \text{key}), S, R), \\ &\quad\ \text{snd-proof(key}, S, R) \end{aligned}$$

Because both S-proof and R-proof are complete when execution of the protocol terminates (after executing the five protocol steps), the protocol is correct.

In verifying the correctness of the two-phase protocol 3, we assumed that the three proof sets of this protocol are empty when execution of the protocol starts. However, this assumption cannot be enforced (and in fact can be violated) in practice.

For example, consider the case where this protocol is executed twice: In the first execution, S sends encrypt($x1$, key) and key to R, and in the second execution, S sends encrypt($x2$, key) and key to R. Note that S sends different texts but sends the same key in these two executions.

After the first execution of this protocol terminates, the S-proof set

and the R-proof set are as follows:

$$\begin{aligned} S\text{-proof} &= \text{rcv-proof}(\text{encrypt}(x1, \text{key}), S, R), \\ &\quad\ \ \text{rcv-proof}(\text{key}, S, T) \\ R\text{-proof} &= \text{snd-proof}(\text{encrypt}(x1, \text{key}), S, R), \\ &\quad\ \ \text{snd-proof}(\text{key}, S, R) \end{aligned}$$

Assume that before the second execution of this protocol starts, the S-proof set is allowed to keep the message rcv-proof(key, S, T) and the R-proof set becomes empty, as usual. Now, the second execution of the protocol starts by S sending the message snd-proof(encrypt($x2$, key), S, R) to R and receiving back the message rcv-proof(encrypt($x2$, key), S, R) from R. At this point, the S-proof set and the R-proof set of the protocol become as follows:

$$\begin{aligned} S\text{-proof} &= \text{rcv-proof}(\text{key}, S, T), \\ &\quad\ \ \text{rcv-proof}(\text{encrypt}(x2, \text{key}), S, R) \\ R\text{-proof} &= \text{snd-proof}(\text{encrypt}(x2, \text{key}), S, R) \end{aligned}$$

Note that at this point, the S-proof set is complete but the R-proof set is not. Thus, executing the third step of the protocol causes the second execution of the protocol to terminate yielding a complete S-proof set and an incomplete R-proof set. Therefore this protocol is incorrect.

The fact that this protocol is incorrect is caused by two factors. First, when the second execution of the protocol starts, the S-proof set is allowed to keep a message that was sent during the first execution of the protocol. Second, S is allowed to use the same key in two distinct executions of the protocol. Both these factors need to be allowed in order to make the protocol

incorrect. Therefore, to make the protocol correct, we need to disallow either one of these two factors. Next we describe how to design keys such that S is compelled to use distinct keys in distinct executions of the protocol.

Consider any execution of two-phase protocol 3, where some text x is to be sent from party S to party R via party T. In this execution of the protocol, we require S to use a key that is the result of applying a secure hash function H to the concatenation of four items: text x, identifier of party S, identifier of party R, and identifier of party T. Thus,

$$\text{key} = H(\text{text } x \mid \text{id of } S \mid \text{id of } R \mid \text{id of } T)$$

where "|" denotes the concatenation operator. Note that this design of the key ensures that S will generate distinct keys to be used in distinct execution of the protocol.

In the third step of this protocol execution, S sends two messages rcv-proof(encrypt(x, key), S, R) and snd-proof(key, S, R) to T. Then T performs two checks to ensure that these messages are correct. First, T checks that the key in the second message can indeed decrypt the encrypted text x in the first message. Second, T checks that the key in the second message has the right structure described above.

Next, we describe two-phase protocol 4, which enables party S to send the same text x to n parties, denoted $R.1, \ldots, R.n$, via trusted third party T.

Two-phase Protocol 4:

Consider the following two-phase multiparty protocol that consists of ($3n$ +

2) steps.

$S \rightarrow R.1$: snd-proof(encrypt(x, key), S, $R.1$)

...

$S \rightarrow R.n$: snd-proof(encrypt(x, key), S, $R.n$)

$S \leftarrow R.1$: rcv-proof(encrypt(x, key), S, $R.1$)

...

$S \leftarrow R.n$: rcv-proof(encrypt(x, key), S, $R.n$)

$S \rightarrow T$: rcv-proof(encrypt(x, key), S, $R.1$),
snd-proof(key, S, $R.1$),
...
...
rcv-proof(encrypt(x, key), S, $R.n$),
snd-proof(key, S, $R.n$)

$S \leftarrow T$: rcv-proof(key, S, T)

$R.1 \leftarrow T$: snd-proof(key, S, $R.1$)

...

$R.n \leftarrow T$: snd-proof(key, S, $R.n$)

Note that this protocol has (n+2) proof sets, namely S-proof, $R.1$-proof, ..., $R.n$-proof, and T-proof.

The S-proof set in this protocol is complete iff this set contains the following (n + 1) messages:

rcv-proof(encrypt(x, key), S, $R.1$),

...

rcv-proof(encrypt(x, key), S, $R.n$),

rcv-proof(key, S, T)

Also each $R.i$-proof set in this protocol is complete iff this set contains the two messages: snd-proof(encrypt(x, key), S, $R.i$) and snd-proof(key, S, $R.i$).

Also note that the key in this protocol is structured as follows:

key = H(text x | id of S | id of $R.1$ | ... | id of $R.n$ | id of T)

where "|" is the concatenation operator.

3.7 Security Analysis

In this section, we present a security analysis of two-phase protocol 3 in this chapter. (A similar security analysis can also be presented for two-phase protocol 4 in the chapter.) Protocol 3 is designed to defend against the four security attacks: malicious parties, message loss, collusion attacks, and replay attacks. Next, we describe these four attacks and show that protocol 3 can succeed in defending against all four attacks.

3.7.1 Malicious Parties

By definition, the trusted third party T in protocol 3 is not malicious. This means that party T sends every message that is supposed to send when it is supposed to send it, as specified by the protocol.

On the other hand, either one of the other two parties S and R can be malicious. A malicious party can refrain from sending any message that is supposed to send, as specified by protocol 3, in order to prevent this other party from obtaining its complete proof. However, because the protocol is two-phase, the malicious party cannot refrain from sending any message specified by the protocol until this party has already obtained its own complete proof.

Referring to protocol 3, party S cannot refrain from sending the snd-proof message in the first step and cannot refrain from sending the rcv-proof and snd-proof messages in the third step because party S does not obtain its complete proof until after the fourth step of the protocol. Similarly, party R cannot refrain from sending the rcv-proof message in the second step of the protocol because party R does not obtain its complete proof until after the fifth step of the protocol.

It follows from this discussion that no party in protocol 3 can refrain from sending any message that is supposed to send as specified by the protocol.

3.7.2 Message Loss

Any message that is supposed to be sent by one party and to be received by another party during the execution of protocol 3 can be lost and not received by the other party. There are two possible sources of message loss: an unreliable communication medium between the two parties and maliciousness of the sending party.

We assume that protocol 3 is to be executed on top of a reliable trans-

port service (such as TCP in the internet). This transport service takes any message that is sent by any party during the execution of protocol 3 and transmits this message any number of times until the message is ultimately received by its intended party. Therefore, message loss due to an unreliable communication medium is not possible.

In Section 3.7.1 above, we argued that no party in protocol 3, whether malicious or not, can refrain from sending any message that is supposed to send as specified by the protocol. Therefore, message loss due to the maliciousness of the sending party is also not possible.

3.7.3 Collusion Attacks

Consider an attack scenario where a party S executes protocol 3 to send text x to a party R via a trusted party T. After executing protocol 3, party S obtains its proof which consists of the following two messages:

rcv-proof(encrypt(x, key), S, R)

rcv-proof(key, S, T)

Later party S forwards these two messages to a malicious party S' which attempts to modify these messages in order to obtain a proof that party S' has sent text x to party R via a trusted party T.

However, any attempt of party S' to modify these two messages will fail because the first message is signed by the private key of party R and the second message is signed by the private key of the trusted party T and neither S nor S' has access to these private keys.

3.7.4 Replay Attacks

If party S is malicious, then it will use same key in two distinct executions of protocol 3. Note that if the same key is used in two distinct executions of protocol 3, then some messages that have been generated and used as part of the proofs of one execution can also be used as part of the proofs of a later execution. Thus, these messages need not be generated in the second execution of the protocol.

Assume for example that party S uses the same key in two distinct executions of protocol 3. In the first execution of protocol 3, S uses key to send text x to party $R.1$ via the trusted party T. In the second execution of protocol 3, S uses the same key to send the same text x to another party $R.2$ via the trusted party T.

The proof that S needs to obtain in the first execution of protocol 3 consists of the two messages:

rcv-proof(encrypt(x, key), S, $R.1$)

rcv-proof(key, S, T)

Also the proof that S needs to obtain in the second execution of protocol 3 consists of the two messages:

rcv-proof(encrypt(x, key), S, $R.2$)

rcv-proof(key, S, T)

Therefore, party S obtains the message rcv-proof(key, S, T) in the first execution of protocol 3 and does not need to obtain it in the second execution of protocol 3. Thus, party S needs to execute only the first step of the protocol

during its second execution of protocol 3.

To defend against this attack by a malicious party S, we prevent S from choosing any key that it likes. Instead, we require that any key that party S chooses to send text x to party R via a trusted party T should be the result of applying a secure hash function H to the concatenation of four items: text x, identifier of party S, identifier of party R, and identifier of party T:

$$\text{key} = H(\text{text } x \mid \text{id of } S \mid \text{id of } R \mid \text{id of } T)$$

In this case, the above attack cannot succeed because the key of the first execution of protocol 3, say key.1, will be different from the key of the second execution of protocol 3, say key.2:

$$\text{key.1}=H(\text{text } x \mid \text{id of } S \mid \text{id of } R.1 \mid \text{id of } T)$$

$$\text{key.2}=H(\text{text } x \mid \text{id of } S \mid \text{id of } R.2 \mid \text{id of } T)$$

3.8 Conclusion and Chapter Summary

In this chapter, we identified a rich class of nonrepudiation protocols called two-phase protocols. We discussed how to specify and verify protocols in this class and argued that the specification and verification of a two-phase protocol are simpler than those of a comparable non-two-phase protocol. Our argument is based on three observations. First, executions of two-phase protocols are deterministic. Second, the parties involved in a two-phase protocol do not need to utilize real-time clocks. Third, the messages exchanged in a two-phase protocol need to be of only two types: snd-proof messages and rcv-proof messages.

The two two-phase protocols 3 and 4 are new. In particular, neither of these protocols utilizes session labels that are distinct for distinct execution sessions of the protocol. Instead, each of these two protocols requires the encryption keys to have a special structure which allows these keys to be used as distinct session labels for distinct execution sessions.

(With the notable exception of the non-two-phase protocol in [13] and [53], most non-two-phase protocols utilize session labels that are distinct for distinct execution sessions. Unfortunately, the non-two-phase protocol in [13] and [53] is quite involved. For example, this protocol consists of three phases: main protocol phase, abort phase, and recovery phase. Moreover, the exchanged messages in this protocol are of eight types. And execution of this protocol is highly non-deterministic.)

Chapter 4

Nonrepudiation Protocols without a Trusted Party

The material that is presented in this chapter is based on the paper [3] [1]. In this chapter, we show that nonrepudiation protocols that do not involve a third party can be designed under reasonable assumptions. Moreover, we identify necessary and sufficient (reasonable) assumptions under which these protocols can be designed. Finally, we present the first ever ℓ-nonrepudiation protocol that involves ℓ parties (none of which is trusted), where $\ell \geq 2$.

A nonrepudiation protocol from party S to party R performs two tasks. First, the protocol enables party S to send to party R some text along with sufficient evidence (that can convince a judge) that the text was indeed sent by S to R. Second, the protocol enables party R to receive the sent text from S and to send to S sufficient evidence (that can convince a judge) that the text was indeed received by R from S.

Each nonrepudiation protocol is also required to fulfill the following opportunism requirement. During any execution of the nonrepudiation protocol from S to R, once a party (S or R, respectively) recognizes that it has al-

[1] Muqeet Ali did ninety percent of the work presented in the paper

ready collected all its needed evidence, then this party concludes that it gains nothing by continuing to execute the protocol and so it terminates. The other party (R or S, respectively) continues to execute the protocol with the hope that it will eventually collect all its needed evidence.

The opportunism requirement which is satisfied by each party in a nonrepudiation protocol can be thought of as a failure of that party. It is important to contrast this failure model with the failure model used in the celebrated paper of Cleve [14]. Recall that Cleve's paper states an impossibility result regarding agreement on random bits chosen by two processes provided that one of the processes is faulty. In Cleve's paper the faulty process can fail at any time during the execution of the protocol. Therefore, Cleve's impossibility result is not applicable in our case, as in our model, failures occur only as dictated by the opportunism requirement.

The intuitive reasoning behind our failure model is that parties do not wish to stop executing the protocol (and thus fail) before collecting their needed evidence. Once a party recognizes that it has already collected all its needed evidence, this party decides to stop executing the protocol (i.e., fail) because it gains nothing by continuing to execute the protocol.

The opportunism requirement, that needs to be fulfilled by each nonrepudiation protocol, makes the task of designing nonrepudiation protocols very hard. This is because once a party in a nonrepudiation protocol terminates (because it has recognized that it has already collected all its needed evidence), then from where will the other party continue to receive its needed evidence?

The standard answer to this question is to assume that a nonrepudiation protocol from party S to party R involves three parties: the two original parties S and R and a third party T, which is often referred to as a trusted party. Note that the objective of each original party is to collect its own evidence, whereas the objective of the third party is to help the two original parties collect their respective evidence. Therefore, the opportunism requirement for the third party T can be stated as follows. Once T recognizes that the two original parties have already collected their evidence (or are guaranteed to collect their evidence soon), T terminates.

An execution of a nonrepudiation protocol from party S to party R that involves the three parties S, R, and T can proceed in three steps as follows:

1. Party S sends some text to party T which forwards it to party R.
2. Party T computes sufficient evidence to establish that S has sent the text to R then T forwards this evidence to R.
3. Party T computes sufficient evidence to establish that R has received the text from S then T forwards this evidence to S.

Most nonrepudiation protocols that have been published in the literature involve three parties: the two original parties and a third party [62]. A nonrepudiation protocol that does not involve a third party was published in [41]. We refer to this protocol as the MR protocol in reference to its two authors Markowitch and Roggeman. Unfortunately, the correctness of this protocol is questionable as discussed next.

In the MR protocol, party S first sends to party R the text encrypted using a symmetric key SK that only S knows. Then party S sends to party R an arbitrary number of random numbers, each of which looks like, but in fact is quite different from, the symmetric key SK. Finally, party S sends to party R the symmetric key SK. After R receives each of these messages from S, R sends to S an ack message that acknowledges receiving the message from S.

The evidence that R needs to collect is the first message (containing the text encrypted using SK) and the last message (containing SK). The evidence that S needs to collect is the acknowledgements of R receiving the first and last messages.

In the MR protocol, when party R receives the i-th message from S, where i is at least 2, R recognizes that the message content is either a random number or the symmetric key SK. Thus, R can use the message content in an attempt to decrypt the encrypted text (which R has received in the first message). If this attempt fails, then R recognizes that the message content is a random number and proceeds to send an ack message to S. If this attempt succeeds, then R recognizes that the message content is the symmetric key SK and terminates right away (in order to fulfill the opportunism requirement) without sending the expected ack message to S. In this case, R succeeds in collecting all the evidence that it needs while S fails in collecting all the evidence that it needs.

To prevent this problematic scenario, the designers of the MR protocol adopted the following three assumptions: (1) there is a lower bound lb on the

time needed by R to decrypt the encrypted text, (2) S knows an upper bound ub on the round trip delay from S to R and back to S, and (3) lb is larger than ub. Based on these assumptions, if party S sends a random number to party R and does not receive back the expected ack message for at least ub time units, then S recognizes that R has tried to cheat (but failed) by attempting to decrypt the encrypted text using the received random number. In this case, party S aborts executing the protocol and both S and R fail to collect all their needed evidence. Now, if party S sends a random number to party R and receives back the expected ack message within ub time units, then both parties continue to execute the protocol.

Unfortunately, party R can secretly decrease the value of lb, for example by employing a super computer to decrypt the encrypted text, such that assumption (3) above is violated. By violating assumption (3), the attempts of R to cheat can go undetected by party S and the MR protocol can end up in a compromised state where party R has terminated after collecting all its needed evidence but party S is still waiting to collect its needed evidence that will never arrive. This problematic scenario calls into question the correctness of the MR protocol.

In this chapter, we discuss how to design nonrepudiation protocols that do not involve a trusted party. We make the following five contributions:

1. We first state the round-trip assumption as follows: Party R knows an upper bound on the round trip delay from R to S and back to R. Then

we show that adopting this assumption is both necessary and sufficient for designing nonrepudiation protocols from S to R that do not involve a third trusted party and where no sent message is lost.

2. Our sufficiency proof in 1 consists of designing the first (provably correct) nonrepudiation protocol from S to R that does not involve a third party and where no sent message is lost.

3. We state the bounded-loss assumption as follows: Party R knows an upper bound on the number of messages that can be lost during any execution of the protocol. Then we show that adopting both the round-trip assumption and the bounded-loss assumption is both necessary and sufficient in designing nonrepudiation protocols from S to R that do not involve a third trusted party and where every sent message may be lost.

4. Our sufficiency proof in 3 consists of designing the first (provably correct) nonrepudiation protocol from S to R that does not involve a third party and where every sent message may be lost.

5. We extend the nonrepudiation protocol in 2 that involves only two parties, S and R, into an ℓ-nonrepudiation protocol that involves ℓ parties (none of which is trusted), where $\ell \geq 2$.

4.1 Nonrepudiation protocols

In this section, we present our specification of a nonrepudiation protocol that does not involve a third party. A nonrepudiation protocol from party S

to party R that does not involve a third party is a communication protocol between parties S and R that fulfills the following requirements.

(a). **Message Loss:** During each execution of the protocol, each message that is sent by either party (S or R, respectively) is eventually received by the other party (R or S, respectively).

(b). **Message Alternation:** During any execution of the protocol, the two parties S and R exchange a sequence of messages. First, party S sends msg.1 to party R. Then when party R receives msg.1, it sends back msg.2 to party S, and so on. At the end when R receives msg.(r-1), where r is an even integer whose value is at least 2, R sends back msg.r to S. This exchange of messages can be represented as follows:

$S \rightarrow R$: msg.1
$S \leftarrow R$: msg.2
...
$S \rightarrow R$: msg.(r-1)
$S \leftarrow R$: msg.r

(c). **Message Signatures:** Party S has a private key that only S knows and the corresponding public key that all parties know. Party S uses its private key to sign every message before sending this message to R so that S can't later repudiate that it has generated this message. Similarly,

Party R has a private key that only R knows and the corresponding public key that all parties know. Party R uses its private key to sign every message before sending this message to S so that R can't later repudiate that it has generated this message.

(d). **Collected Evidence:** Both parties S and R collect evidence during execution of the protocol. The evidence collected by party S is a subset of those messages received by party S (from party R). Similarly, the evidence collected by party R is a subset of those messages received by party R (from party S).

(e). **Guaranteed Termination:** Every execution of the protocol is guaranteed to terminate in a finite time.

(f). **Termination Requirement:** Party S terminates only after S sends some text to R and only after S receives from R sufficient evidence to establish that R has indeed received the sent text from S. Similarly, party R terminates only after R receives from S both the sent text and sufficient evidence to establish that S has indeed sent this text to R.

(g). **Opportunism Requirement:** If during any execution of the protocol, party S recognizes that it has already sent some text to R and has later received from R sufficient evidence to establish that R has indeed received this text, then S terminates. Similarly, if during any execution of the protocol, party R recognizes that it has received from S some text

and sufficient evidence to establish that S has indeed sent this text, then R terminates.

(h). **Judge:** Anytime after execution of the nonrepudiation protocol terminates, each of the two parties, S or R, can submit its collected evidence to a judge that can decide whether the submitted evidence is valid and should be accepted or it is invalid and should be rejected. The decision of the judge is final and legally binding on both S and R. To help the judge make the right decision, we assume that the judge knows the public keys of S and R. We also assume that the judge has a public key (that both S and R know) and a corresponding private key (that only the judge knows).

(Note that the role of the judge is different than that of a trusted third party. The trusted third party is used to generate and distribute the needed evidence to the two parties, S and R. Therefore, the trusted third party is directly involved during the execution of the nonrepudiation protocol. The judge, however, is not directly involved during the execution of the protocol, and it never generates nor distributes any part of the needed evidence to either party. The judge only verifies the submitted evidence after it has already been collected during execution of the protocol. In fact every nonrepudiation protocol that has been published in the past has both a trusted third party and a judge)

The opportunism requirement needs some explanation. Once party S recognizes that it has already collected sufficient evidence to establish that party R

has indeed received the text from S, S concludes that it gains nothing by continuing to participate in executing the protocol and so S terminates. (In this case, only R may gain by continuing to participate in executing the protocol.)

Similarly, once party R recognizes that it has already collected sufficient evidence to establish that party S has indeed sent the text to R, R concludes that it gains nothing by continuing to participate in executing the protocol and so R terminates.

Next, we state a condition, named the round-trip assumption and show in the next section that adopting this assumption is both necessary and sufficient to design nonrepudiation protocols from party S to party R that do not involve a third party.

Round-Trip Assumption: Party R knows an upper bound t (in time units) on the round trip delay from R to S and back to R

4.2 Necessary and Sufficient Conditions for Nonrepudiation Protocols

We prove that it is necessary to adopt the round-trip assumption when designing nonrepudiation protocols from party S to party R that do not involve a third party.

Theorem 4.1. *In designing a nonrepudiation protocol from party S to party R, that does not involve a third party, it is necessary to adopt the round-trip*

assumption.

Proof. Consider an execution of a nonrepudiation protocol from party S to party R that does not involve a third party. This execution fulfills the eight requirements that are listed in Section 4.1, namely message loss, message alternation, message signature, collected evidence, guaranteed termination, termination requirement, opportunism requirement, and judge.

From the message alternation requirement, the exchanged messages during this execution of the protocol can be represented as follows:

$S \rightarrow R$: msg.1

$S \leftarrow R$: msg.2

...

$S \rightarrow R$: msg.(r-1)

$S \leftarrow R$: msg.r

During this execution of the protocol, when party S receives a msg.i from party R, then by the collected evidence requirement, party S may recognize that msg.i is part of its evidence that needs to be collected and add msg.i to its set of collected evidence. Then party S examines its set of collected evidence and reaches one of two conclusions: either the set of collected evidence is not yet sufficient to establish that R has indeed received the sent text from S, or the set of collected evidence is already sufficient to establish that R has indeed received the sent text from S.

If party S reaches the first conclusion, then S recognizes by the termination requirement that it needs to receive at least one more message from R, and so S goes ahead and sends msg.(i+1) to R then waits to receive msg.(i+2) from R.

On the other hand, if party S reaches the second conclusion, then S recognizes by the opportunism requirement that it needs to terminate. In this case, if R knows no upper bound (in time units) on the round trip delay from R to S and back to R, then R will continue to wait indefinitely for the next message, namely msg.(i+1), which will never arrive, contradicting the guaranteed termination requirement. (Note that if R knows an upper bound t on the round trip delay from R to S and back to R, then R will wait t time units for msg.(i+1) before recognizing that there is no msg.(i+1) and that the current set of evidence collected by R is sufficient to establish that S has indeed sent the text to R.)

Thus, if party S ever reaches the second conclusion when S receives a msg.i from R, then R needs to know an upper bound on the round trip delay from R to S and back to R (in order to avoid contradicting the guaranteed termination requirement). In fact, party S is guaranteed to reach this second conclusion when S receives the last message, namely msg.r, from R. Therefore, the round-trip assumption needs to be adopted during any execution of the protocol. □

Next, we prove that it is sufficient to adopt the round-trip assumption

in order to design a nonrepudiation protocol from party S to party R that does not involve a third party.

Theorem 4.2. *It is sufficient to adopt the round-trip assumption in order to design a nonrepudiation protocol from party S to party R that does not involve a third party.*

Proof. We present a design of a nonrepudiation protocol from party S to party R that does not involve a third party and show that correctness of this protocol is based on adopting the round-trip assumption. Our presentation of this protocol consists of four steps. In each step, we start with a version of the protocol then show that this version is incorrect (by showing that it violates one of the requirements in Section 4.1). We then proceed to modify this protocol version in an attempt to make it correct. After four steps, we end up with a correct nonrepudiation protocol (that satisfies all eight requirements in Section 4.1).

First Protocol Version: In this protocol version, party S sends a txt message to party R which replies by sending back an ack message. The exchange of messages in this protocol version can be represented as follows.

$S \rightarrow R$: txt
$S \leftarrow R$: ack

The txt message contains: (1) the message sender S and receiver R, (2) the text that S needs to send to R, and (3) signature of the message using the private key of the message sender S. Similarly, the ack message contains: (1) the message sender R and receiver S, (2) the text that S needs to send to R, and (3) signature of the message using the private key of the message sender R.

The txt message is the evidence that R needs to collect and can later present to the judge to get the judge to declare that the text in the message was indeed sent by S to R. Similarly, the ack message is the evidence that S needs to collect and can later present to the judge to get the judge to declare that the text in the message was indeed received by R from S.

This protocol version is incorrect for the following reason. When R receives the txt message, it recognizes that it has already collected sufficient evidence to establish that S has indeed sent the text to R and so R terminates, by the opportunism requirement, before it sends the ack message to S. Party S ends up waiting indefinitely for the ack message that will never arrive violating the guaranteed termination requirement.

To make this protocol version correct, we need to devise a technique by which R does not recognize that it has collected sufficient evidence to establish that S has indeed sent the text to R, even after R has collected such evidence.

Second Protocol Version: In this protocol version, party S sends n txt messages to party R, where n is a positive integer selected at random by S

and is kept as a secret from R. The exchange of messages in this protocol version can be represented as follows.

$S \rightarrow R$: txt.1

$S \leftarrow R$: ack.1

...

$S \rightarrow R$: txt.n

$S \leftarrow R$: ack.n

Each txt.i message contains: (1) the message sender S and receiver R, (2) the text that S needs to send to R, (3) the sequence number of the message i, and (4) signature of the message using the private key of the message sender S. Similarly, each ack.i message contains: (1) the message sender R and receiver S, (2) the text that S needs to send to R, (3) the sequence number of the message i, and (4) signature of the message using the private key of the message sender R.

The txt.n message is the evidence that R needs to collect and can later present to the judge to get the judge to declare that the text in the message was indeed sent by S to R. Similarly, the ack.n message is the evidence that S needs to collect and can later present to the judge to get the judge to declare that the text in the message was indeed received by R from S.

When R receives the txt.n message from S, then (because R does not know the value of n) R does not recognize that it has just received sufficient

evidence to establish that S has indeed sent the text to R. Thus, R does not terminate and instead proceeds to send the ack.n message to S.

When S receives the ack.n message from R, then (because S knows the value of n) S recognizes that it has just received sufficient evidence to establish that R has received the text sent from S to R. Thus, by the opportunism requirement, S terminates and R ends up waiting indefinitely for the txt.(n+1) message that will never arrive violating the guaranteed termination requirement.

This protocol version is incorrect. To make it correct, we need to devise a technique by which party R recognizes, after S terminates, that R has already collected sufficient evidence to establish that S has indeed sent the text to R.

Third Protocol Version: This protocol version is designed by modifying the second protocol version (discussed above) taking into account the adopted round-trip assumption, namely that R knows an upper bound t (in time units) on the round trip delay from R to S and back to R.

The exchange of messages in the third protocol version is the same as that in the second protocol version. However, in the third protocol version, every time R sends an ack.i message to S, R activates a time-out to expire after t time units.

If R receives the next txt.(i+1) before the activated time-out expires,

then R cancels the timeout. If the activated time-out expires before R receives the next txt.(i+1) message, then R recognizes that the last received txt.i message is in fact the txt.n message and so R recognizes that it has already collected sufficient evidence to establish that S has already sent the text to R and so R terminates (by the opportunism requirement). Execution of this protocol version can be represented as follows:

$S \rightarrow R$: txt.1

$S \leftarrow R$: ack.1; R activates time-out

$S \rightarrow R$: txt.2; R cancels time-out

...

$S \rightarrow R$: txt.n; R cancels time-out

$S \leftarrow R$: ack.n; R activates time-out; S terminates

time-out expires; R terminates

This protocol version still has a problem. After execution of the protocol terminates, party R may decide to submit its collected evidence, namely the txt.n message, to the judge so that the judge can certify that S has indeed sent the text in the txt.n message. The judge can make this certification if it observes that the sequence number of the txt.n message equals the random integer n that S selected when execution of the protocol started. Unfortunately, the value of n is not included in the txt.n message.

Similarly, after execution of the protocol terminates, party S may de-

cide to submit its collected evidence, namely the ack.n message, to the judge so that the judge can certify that R has indeed received the text in the ack.n message. The judge can make this certification if it observes that the sequence number of the ack.n message equals the random integer n that S selected when execution of the protocol started. Unfortunately, the value of n is not included in the ack.n message.

To solve these two problems, we need to devise a technique by which the value of n is included in every txt.i message and every ack.i message such that the following two conditions hold. First, the judge can extract the value of n from any txt.i or ack.i message. Second, party R can't extract the value of n from any txt.i message nor from any ack.i message.

Fourth Protocol Version: In this protocol version, two more fields are added to each txt.i message and each ack.i message. The first field stores the encryption of n using a symmetric key KE that is generated by party S and is kept as a secret from party R. The second field stores the encryption of the symmetric key KE using the public key of the judge.

These two fields are computed by party S when execution of the protocol starts and are included in every txt.i message before this message is sent from S to R. When party R receives a txt.i message from party S, party R copies these two fields from the received txt.i message into the next ack.i message before this message is sent from R to S.

After execution of this protocol version terminates, party S can submit

its collected evidence, namely the last ack.i message that S has received from R, to the judge so that the judge can examine the ack.i message and certify that R has received the text in this message from S. The judge makes this certification by checking, among other things, that the sequence number i of the ack.i message is greater than or equal to n. The judge then forwards its certification to party S.

Similarly, after execution of this protocol version terminates, party R can submit its collected evidence, namely the last txt.i message that R has received from S, to the judge so that the judge can examine the txt.i message and certify that S has sent the text in this message to R. The judge makes this certification by checking, among other things, that the sequence number i of the txt.i message is greater than or equal to n. The judge then forwards its certification to party R.

Note that the judge can certify at most one ack.i message from party S, and at most one txt.i message from party R. This restriction forces S to send to the judge only the last ack.i message that S has received from R. This restriction also forces R to send to the judge only the last txt.i message that R has received from S. □

4.3 Nonrepudiation Protocols with Message Loss

In the remainder of this chapter, we consider a richer class of nonrepudiation protocols where sent messages may be lost before they are received. Each protocol in this class is required to fulfill the following requirements.

(a). **Message Loss:** During each execution of the protocol, each message that is sent by either party (S or R, respectively) can be lost before it is received by the other party (R or S, respectively).

(b). **Message Alternation:** During any execution of the protocol where no sent message is lost, the two parties S and R exchange a sequence of messages. First, party S sends msg.1 to party R. Then when party R receives msg.1, it sends back msg.2 to party S, and so on. At the end when R receives msg.(r-1), where r is an even integer whose value is at least 2, R sends back msg.r to S. This exchange of messages can be represented as follows:

$S \rightarrow R$: msg.1

$S \leftarrow R$: msg.2

...

$S \rightarrow R$: msg.(r-1)

$S \leftarrow R$: msg.r

(c). **Message Signatures:** This requirement is the same as the message signature requirement in Section 4.1.

(d). **Collected Evidence:** This requirement is the same as the collected evidence requirement in Section 4.1.

(e). **Guaranteed Termination:** This requirement is the same as the guaranteed termination requirement in Section 4.1.

(f). **Termination Requirement:** This requirement is the same as the termination requirement in Section 4.1.

(g). **Opportunism Requirement:** This requirement is the same as the opportunism requirement in Section 4.1.

(h). **Judge:** This requirement is the same as the requirement of the judge in Section 4.1.

Next, we state a condition, named the bounded-loss assumption. We then show in the next section that adopting this assumption along with the round-trip assumption (stated in Section 4.1) is both necessary and sufficient to design nonrepudiation protocols from party S to Party R that do not involve a third party and where sent messages can be lost.

Bounded-Loss Assumption: Party R knows an upper bound K on the number of messages that can be lost during any execution of the nonrepudiation protocols

4.4 Necessary and Sufficient Conditions for Nonrepudiation Protocols with Message Loss

We prove that it is necessary to adopt both the round-trip assumption and the bounded-loss assumption when designing nonrepudiation protocols

from party S to party R that do not involve a third party and where sent messages may be lost.

Theorem 4.3. *In designing a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost, it is necessary to adopt the round-trip assumption.*

Proof. Consider a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost. Consider an execution of this protocol where no message is lost. This execution fulfills the eight requirements that are listed in Section 4.4, namely message loss, message alternation, message signature, collected evidence, guaranteed termination, termination requirement, opportunism requirement, and judge.

From the message alternation requirement, the exchanged messages during this execution of the protocol can be represented as follows:

$S \rightarrow R$: msg.1
$S \leftarrow R$: msg.2
...
$S \rightarrow R$: msg.(r-1)
$S \leftarrow R$: msg.r

This execution of the nonrepudiation protocol with message loss is the same as the execution in the proof of Theorem 4.1, which states that in design-

ing a nonrepudiation protocol (with no message loss) from party S to party R, that does not involve a third party, it is necessary to adopt the round-trip assumption. As the execution of the nonrepudiation protocol discussed in this proof is the same as in Theorem 4.1, the proof of this theorem follows from the reasoning given in Theorem 4.1. Therefore, it is necessary to adopt the round-trip assumption in designing a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost. □

Theorem 4.4. *In designing a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost, it is necessary to adopt the bounded-loss assumption.*

Proof. Consider a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost. Also consider an execution of this protocol where only msg.r is sent and lost several times. (Recall that from Theoem 4.3, party R knows an upper bound t in time units on the round trip delay from R to S and back to R.)

From the message alternation requirement, the exchanged messages during this execution of the protocol can be represented as follows:

$S \rightarrow R$: msg.1

$S \leftarrow R$: msg.2

...

$S \rightarrow R$: msg.(r-1)

$S \leftarrow R$: msg.r

$S \leftarrow R$: msg.r (after R waits for but does not receive msg.(r+1) for t time units)

$S \leftarrow R$: msg.r (after R waits for but does not receive msg.(r+1) for t time units)

During this execution of the protocol, when party S receives msg.i from party R, then by the collected evidence requirement, party S may decide to add msg.i to its set of collected evidence. Then party S examines its set of collected evidence and reaches one of two conclusions: either the set of collected evidence is not yet sufficient to establish that R has indeed received the sent text from S, or the set of collected evidence is already sufficient to establish that R has indeed received the sent text from S.

If party S reaches the first conclusion, then S recognizes by the termination requirement that it needs to receive at least one more message from R, and so S goes ahead and sends msg.(i+1) to R then waits to receive msg.(i+2) from R.

If party S reaches the second conclusion, then S recognizes by the opportunism requirement that the value of i is in fact r and so S terminates while R continues to wait for msg.(i+1) that will never arrive. After waiting for t time units without receiving msg.(i+1), R recognizes that either msg.i or msg.(i+1) was lost and so R times-out, resends msg.i to S, and continues to

wait to receive msg.(i+1) that will never arrive, and the cycle repeats. There are two cases to be considered.

In the first case, R knows no upper bound on the number of messages that can be lost during any execution of the protocol. In this case, R will continue to execute the following two steps indefinitely: First, R waits for t time units to receive msg.(r+1) that will never arrive. Second, R times-out after t time units and resends msg.r. In this case R never terminates which contradicts the guaranteed termination requirement.

In the second case, R knows an upper bound K on the number of messages that can be lost during any execution. In this case, R executes the above two steps K times only then recognizes that its current set of collected evidence is sufficient to establish that S has indeed sent the text to R and so R terminates by the opportunism requirement.

This discussion necessitates that R knows an upper bound K on the number of messages that can be lost during any execution of the protocol. Therefore, the bounded-loss assumption needs to be adopted during any execution of the protocol □

Next, we prove that it is sufficient to adopt both the round-trip assumption and the bounded-loss assumption in order to design a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost.

Theorem 4.5. *It is sufficient to adopt both the round-trip assumption and the bounded-loss assumption in order to design a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost.*

Proof. In our proof of Theorem 4.2, we adopted the round-trip assumption in order to design a nonrepudiation protocol, which we refer to in this proof as protocol P, from party S to party R that does not involve a third party and where no sent message is ever lost. In the current proof, we adopt the bounded-loss assumption in order to modify protocol P into protocol Q, which is a nonrepudiation protocol from party S to party R that does not involve a third party and where sent messages may be lost.

In protocol P, every time party R sends an ack.i message to party S, R activates a time-out to expire after t time units, where (by the round-trip assumption) t is an upper bound on the round trip delay from R to S and back to R. Because no sent message is ever lost in protocol P, then if the activated time-out expires before R receives the next txt.(i+1) message from S, R concludes that S has already terminated, the next txt.(i+1) message will never arrive, and R has already collected sufficient evidence to establish that S has sent the text to R. In this case, R also terminates fulfilling the opportunism requirement.

In protocol Q, every time party R sends an ack.i message to party S, R activates a time-out to expire after t time units. However, because every sent

message in protocol Q may be lost, if the activated time-out expires before R receives the next txt.(i+1) message from S, then R concludes that either the ack.i message or the txt.(i+1) message is lost, and in this case R sends the ack.i message once more to S and activates a new time-out to expire after t time units, and the cycle repeats.

By the bounded-loss assumption, R knows an upper bound K on the number of messages that can be lost during any execution of protocol Q. Therefore, the cycle of R sending an ack.i message then the activated time-out expiring after t time units can be repeated at most K times.

If the activated time-out expires for the (K+1)-th time, then R concludes that S has already terminated, the next txt.(i+1) message will never arrive, and R has already collected sufficient evidence to establish that S has sent the text to R. In this case, R terminates fulfilling the opportunism requirement. □

4.5 An ℓ-Nonrepudiation Protocol

In the Proof of Theorem 4.2, we presented a nonrepudiation protocol from party S to party R that does not involve a trusted party and where no sent message is lost. In this section, we discuss how to extend this protocol to a nonrepudiation protocol that involves ℓ parties, namely $P1, \cdots, P\ell$, and satisfies three conditions: (1) $\ell \geq 2$, (2) none of the involved parties in the protocol is a trusted party, and (3) no sent message during any execution of the protocol is lost. We refer to this extended protocol as an ℓ-nonrepudiation

protocol.

The objectives of an ℓ-nonrepudiation protocol are as follows. For each two parties in the protocol, say Pi and Pj, the protocol achieves two objectives:

1. One of the two parties, say Pi, is enabled to send the text to the other party Pj and to receive from Pj sufficient evidence that can convince a judge that Pj has indeed received the text from Pi.

2. The other party Pj is enabled to receive the text from Pi and to receive sufficient evidence from Pi that can convince a judge that Pi has indeed sent the text to Pj.

Therefore, each party Pi ends up collecting sufficient evidence from every other party Pj indicating that either Pj has indeed received the text from Pi or Pj has indeed sent the text to Pi.

Before we describe our ℓ-nonrepudiation protocol, as an extension of the 2-nonrepudiation protocol in the proof of Theorem 4.2, we need to introduce two useful concepts: parent and child of a party Pi. Each of the parties $P1$, $\cdots$, $P(i-1)$ is called a parent of party Pi. Also each of the parties $P(i+1)$, $\cdots$, $P\ell$ is called a child of party Pi. Note that party $P1$ has no parents and parent $P\ell$ has no children. Note also that the number of parents plus the number of children for each party is (ℓ-1).
Execution of our ℓ-nonrepudiation protocol proceeds as follows:

1. Party $P1$ starts by sending a txt.1 message to each one of its children.

2. When a party Pi, where $i \neq 1$ and $i \neq \ell$, receives a txt.1 message from each one of its parents, party Pi sends a txt.1 message to each one of its children.

3. When party $P\ell$ receives a txt.1 message from each one of its parents, party $P\ell$ sends back an ack.1 message to each one of its parents.

4. When a party Pi, where $i \neq 1$ and $i \neq \ell$, receives an ack.1 message from each one of its children, party Pi sends an ack.1 message to each one of its parents.

5. When party $P1$ receives an ack.1 message from each one of its children, party $P1$ sends a txt.2 message to each one of its children, and the cycle consisting of Steps 2, 3, 4, and 5 is repeated n times until $P1$ receives an ack.n message from each one of its children. In this case, party $P1$ collects as evidence the ack.n messages that $P1$ has received from all its children, then $P1$ terminates.

6. Each party Pi, where $i \neq 1$, waits to receive a txt.(n+1) message (that will never arrive) from each one of its parents, then times out after (i-1)*T time units, collects as evidence the txt.n messages that Pi has received from all its parents and the ack.n messages that Pi has received from all its children, then Pi terminates.

Note that T is an upper bound on the round trip delay from any party Pi to any other party Pj and back to Pi. It is assumed that each party, other than $P1$, knows this upper bound T.

4.6 Conclusion and Chapter Summary

In this chapter, we address several problems concerning the design of nonrepudiation protocols from a party S to a party R that do not involve a trusted third party. In such a protocol, S sends to R some text x along with sufficient evidence to establish that S is the party that sent x to R, and R sends to S sufficient evidence that R is the party that received x from S.

Designing such a protocol is not an easy task because the protocol is required to fulfill the following opportunism requirement. During any execution of the protocol, once a party recognizes that it has received its sufficient evidence from the other party, this party terminates right away without sending any message to the other party. In this case, the other party needs to obtain its evidence without the help of the first party. (To fulfill this opportunism requirement, most published nonrepudiation protocols involve a trusted third party T so that when one of the two original parties recognizes that it has already received its sufficient evidence and terminates, the other party can still receive its evidence from T.)

Our main result in this chapter is the identification of two simple conditions that are both necessary and sufficient for designing nonrepudiation protocols that do not involve a trusted third party. .

In proving that these two conditions are sufficient for designing nonrepudiation protocols, we presented an elegant nonrepudiation protocol that is based on the following novel idea. By the time party S recognizes that it has received its evidence, party S has already sent to R its evidence, but R has not yet recognized that it has received all its evidence. In this case, S terminates as dictated by the opportunism requirement but R continues to wait for the rest of its evidence from S. Eventually R times-out and recognizes that S will not send any more evidence. This can only mean that party S has terminated after it has already sent all the evidence to R. Thus, R terminates as dictated by the opportunism requirement.

Chapter 5

Nonrepudiation Protocols in Cloud Systems

In this chapter, we present a nonrepudiation protocol from party S to party R that transfers a file F from party S to party R such that all communications between parties S and R are carried out through a cloud C. A nonrepudiation protocol from party S to party R that is implemented in cloud C ensures that party S gets a proof that a file F was indeed received by party R from party S. It also ensures that party R receives both the file F and the proof that the file F was indeed sent by party S to party R.

In previous nonrepudiation protocols, parties S and R could communicate directly with each other. In this chapter, we introduce nonrepudiation protocols in which the communication is carried out only through a cloud C. This different mode of communication leads us to design a new nonrepudiation protocol from S to R over a cloud C. Throughout this chapter we will assume that the cloud C serves as a permanent storage for the items stored in it, i.e. we assume that cloud C is able to store the file F and the needed proofs by the two parties S and R permanently. Once the file F and two proofs are stored on the cloud C, they can be received at any later time by any of the two parties S and R. Therefore, party S or party R need not store the file F

and the needed proofs in their local storage. These parties can receive these items from the cloud C whenever needed.

In this chapter, we describe the details of the nonrepudiation protocol from party S to party R that is implemented in cloud systems. In Section 5.1, we will present an overview of the protocol. In Section 5.2, we will present the specification of the protocol, and Section 5.3 includes the verification of the proposed protocol. We present the advantages of the protocol in Section 5.4, and conclude in Section 5.5.

5.1 Protocol Overview

The protocol involves three parties: party S, party R, and party C. The protocol enables the transfer of a file F from party S to party R such that all communications are carried out through a cloud C. The protocol will enable party S to send to party R the file F and a proof that the file F was indeed sent by S to R. It also enables party R to receive file F from party S and to send to S a proof that file F was indeed received by R from S. Since all the communications are carried out through a cloud C, it is convenient to replace the proof that file F was sent by party S to party R by the following two proofs:

> Proof1 is a proof that file F was sent by sender S to cloud C
>
> Proof2 is a proof that F was received by cloud C from sender S, and was later sent from C to receiver R

Proof1 and Proof2 together form the proof that file F was sent by party S to

party R.

Similarly, the proof that file F was received by party R from party S can be replaced by the following two proofs:

Proof2 is a proof that F was received by cloud C from sender S, and was later sent from C to receiver R

Proof3 is a proof that F was received by receiver R from cloud C

Proof2 and Proof3 together form the proof that file F was indeed received by party R from party S.

The nonrepudiation protocol from S to R over a cloud C operates in two modes. Next, we give an overview of these two modes of the protocol. The first mode of the protocol has three steps. In the first step, party S sends the file F alongwith Proof1 to party C. In the second step, party C sends Proof2 to party R which serves as an announcement that party C received file F from party S. In the third step, party R sends Proof3 to party C which serves as an acknowledgement of the announcement. At the end of this mode of the protocol, party C ends up storing the file F, and the three proofs: Proof1, Proof2, and Proof3.

In the second mode, party R asks for the file F and its needed proofs from party C, and after party C receives this request from R, party C sends the file F along with Proof1 and Proof2 to party R. Similarly, in the second mode, party S asks for the file F and its needed proofs from party C, after party C receives this request from S, party C sends the file F along with Proof2 and Proof3 to party S

It is noteworthy to mention that the first mode of the protocol is executed only once, but the second mode of the protocol can be executed any number of times. Whenever a party S or R wants to receive the file F and its needed proofs, it can execute the second mode of the protocol. Since the cloud stores the file F and the three proofs permanently, therefore the cloud C can always send both the file F and the needed proofs to each of the two parties S and R.

5.2 Protocol Specification

We assume that each party (i.e., party C, party S, and party R) has a private and public key. Each party knows its private and public keys, and knows the public keys of all other parties. In this section we will describe the steps of the protocol, and the proofs mentioned in the previous section in greater detail. The Proof1 which is a proof that F was sent by party S to party C can be represented as follows:

Proof1: (S,C,R,$H(F)$,signature by S)

where S is the identity of party S

C is the identity of party C

R is the identity of party R

$H(F)$ is a hash of file F using a secure hash function H

the signature of S is computed on the tuple (S,C,R,$H(F)$) using the private key of S

Proof2 which is a proof that file F was received by C from S, and was later sent by C to R can be represented as follows:

Proof2: (S,C,R,$H(F)$,signature by C)

where S is the identity of party S

C is the identity of party C

R is the identity of party R

$H(F)$ is a hash of file F using a secure hash function H

the signature of C is computed on the tuple (S,C,R,$H(F)$) using the private key of C

Proof3 which is a proof that file F was received by party R from party C can be represented as follows:

Proof3: (S,C,R,$H(F)$,signature by R)

where S is the identity of party S

C is the identity of party C

R is the identity of party R

$H(F)$ is a hash of file F using a secure hash function H

the signature of R is computed on the tuple (S,C,R,$H(F)$) using the private key of R

Next we describe the two modes of the protocol. The first mode has three steps which are described below:

1. Store-File Step: In this step, party S sends a file F along with Proof1 to party C. This step can be represented as:

$S \rightarrow C$: (F, Proof1)

2. Announce-File Step: In this step, party C announces to party R that it has received file F from party S. This step can be represented as:

$R \leftarrow C$: (Proof2)

3. Announcement-Received Step: In this step party R sends Proof3 to party C which proves to party C that R has received the announcement sent in the Announce-File Step. This step is represented as:

$R \rightarrow C$: (Proof3)

At the end of the execution of the first mode of the protocol, party C ends up with the following five data items:

F

$H(F)$

Proof1

Proof2

Proof3

These five data items are stored permanently by the cloud C. On the other hand each of the two parties S and R end up with only one item, namely

$H(F)$.

Next, we describe the second mode of the protocol:

In this mode,party R asks for the file F whose hash is $H(F)$ and the two associated proofs Proof1 and Proof2 from party C. After party C receives this request from party R, it sends to party R the file F and its two needed proofs . This step is represented as follows:

$R \rightarrow C$: send me file F whose hash is $H(F)$ and the two associated proofs, Proof1 and Proof2

$R \leftarrow C$: (F,Proof1,Proof2)

Also in the second mode,party S asks for the file F whose hash is $H(F)$ and the two associated proofs Proof2 and Proof3 from party C. After party C receives this request from party S, it sends to party S the file F and its two needed proofs . This step is represented as follows:

$S \rightarrow C$: send me file F whose hash is $H(F)$ and the two associated proofs, Proof2 and Proof3

$S \leftarrow C$: (F,Proof2,Proof3)

At the end of the execution of the second mode of the protocol, party R ends up with file F and the two proofs: Proof1 and Proof2, which together form the proof that file F was sent by party S to party R. Similarly, party S ends up with file F and the two proofs: Proof2 and Proof3, which together

form the proof that file F was indeed received by party R from party S. The second mode of the protocol can be executed any number of times by either of the two parties S and R as the cloud C is able to store the file F and the three proofs permanently.

5.3 Protocol Verification

In this section, we verify the correctness of the nonrepudiation protocol presented in the previous section. We execute the two modes of the protocol. After the execution of the second mode of the protocol, we show that each party ends up with both the file F and its needed proofs. Below we execute the steps of the first mode of the protocol.

After the execution of the first step, party S stores $H(F)$ and party C stores $H(F)$, F and Proof1.

After the execution of the second step, party R stores $H(F)$, and party C stores Proof2

After the execution of the third step, party C stores Proof3

At the end of the execution of the above three steps, party S ends up storing $H(F)$, party R ends up storing $H(F)$, and party C ends up storing the five data items: $H(F)$, F, Proof1, Proof2, and Proof3. This means party C ends up storing the file F and all the proofs.

After the execution of the second mode of the protocol, party R ends up with file F and two proofs Proof1 and Proof2. Also, party S ends up with file F and the two proofs Proof2 and Proof3.

In this way after the end of the execution of the two modes of the protocol. Party S ends up with the file F and the proof that file F was indeed received by R from S. Also, party R ends up with file F and the proof that file F was indeed sent by S to R.

5.4 Protocol Advantages

In this section, we describe the advantage of nonrepudiation protocols in cloud systems as compared to nonrepudiation protocols that are not implemented in cloud systems. In previous nonrepudiation protocols, a party (S or R) needed to store both the file F and its needed proof. For example, party S needed to store both the file F and the proof that F was indeed received by R from S as both these items are needed by the judge in order to be convinced that R indeed received file F from S. Similarly, party R needed to store both the file F and the proof that file F was indeed sent by S to R in order to convince the judge that file F was indeed sent by S to R. Therefore, in these previous nonrepudiation protocols, if any party S or R is not able to store one of the two required items, then it would no longer be able to convince the judge about its claim. However, in the nonrepudiation protocols presented in this chapter, there is no requirement that a party S or R needs to store both the file F and the respective proof. Only the cloud C stores the file F and the proofs indefinitely.

5.5 Conclusion and Chapter Summary

In this chapter, we have presented a new nonrepudiation protocol from party S to party R that is implemented in a cloud C. This protocol is designed to work such that all communications are carried out through a cloud C. The presented protocol works in two modes. The first mode of the protocol is designed such that at the end of its execution, party C is able to store the file F, and the proofs needed by the two parties S and R. In the second mode of the protocol, each party S or R asks for the file F and the needed proofs from party C. After party C receives such a request from a party S or R, Party C sends both the file F and the needed proofs to the respective party. The protocol presented in this chapter has an interesting property which does not require parties S or R to store both the file F and needed proofs. Previously, nonrepudiation protocols required that file F and the needed proofs be stored indefinitely as at any later time a party S or R may need these items in order to convince a judge about its claim.

Chapter 6

A Stabilizing Nonrepudiation Protocol Over a Cloud

In Chapter 5 we presented a nonrepudiation protocol from party S to party R that is implemented in cloud C. In this protocol we assume that cloud C is able to store file F and the needed proofs by the two parties, S and R, for an indefinite period of time. Since file F and the proofs are stored indefinitely in cloud C, parties S and R, do not need to store file F and the needed proofs. Whenever a party, S or R, needs file F and its needed proofs, this party just asks for these items from cloud C through the two actions in the second mode of the protocol.

From Chapter 5, cloud C stores five data items for each file F that is sent from party S to party R: $H(F)$, F, Proof1, Proof2, and Proof3. In the first action in the second mode of the protocol, party S asks for file F, and the two proofs: Proof2 and Proof3. Similarly, in the second action in the second mode, party R asks for file F and the two proofs: Proof1 and Proof2. When cloud C receives a request from party S, it sends file F alongside Proof2 and Proof3 to party S. Similarly, when cloud C receives a request from party R, it sends file F alongside Proof1 and Proof2 to party R.

In this chapter, we consider a scenario where some of the proofs that are stored in cloud C get lost. When some of the proofs stored in cloud C are lost, then it is possible that one party (say S) is able to receive its two needed proofs, but the other party R is not able to receive its two needed proofs. Such a scenario is problematic since in this case the nonrepudiation requirements of the protocol are violated. For example let us consider the scenario where Proof1 stored in cloud C is lost but the other two proofs Proof2 and Proof3 are still present in cloud C. In this case, party S will be able to receive its two needed proofs, Proof2 and Proof3 but party R will not be able to receive its two needed proofs since Proof1 is lost. Similarly, when Proof3 stored in cloud C is lost but the two proofs, Proof1 and Proof2 are still present in cloud C, then party R will be able to receive its two needed proofs but party S will not be able to receive its two needed proofs since Proof3 is lost.

In this chapter, we design a new nonrepudiation protocol from party S to party R over a cloud C which ensures that if one of the two proofs (say Proof1) gets lost but the other proof (say Proof3) is still present in cloud C, then eventually, no party S or R is able to receive its needed proofs from cloud C. Since no party S or R will be able to receive its needed proofs, therefore, the nonrepudiation requirements are not violated.

In this chapter, we make the following contributions:

- We consider the nonrepudiation protocol from S to R over cloud C which is presented in chapter 5, and show that this protocol suffers from a

compromised state when one of the two proofs, Proof1 or Proof3, is lost but the other proof is still present in cloud C. In this compromised state of the protocol one party is able to receive its needed proofs but the other party is not able to receive its needed proofs violating the nonrepudiation requirements of the protocol.

- We design a new nonrepudiation protocol from S to R over cloud C which has an interesting property. This property ensures that if the protocol ever reaches a compromised state where one of the two parties S and R is able to receive its needed proofs but the other party is not able to receive its needed proofs, then after some finite time period, this compromised state changes to a state where no party is able to receive its needed proofs from cloud C. This ensures that the nonrepudiation requirement of the protocol is not violated.

In Section 6.1, we discuss the possible values that a proof stored in cloud C can take. In Section 6.2 we define the state of the protocol. In Section 6.3 we introduce legitimate and illegitimate states of the protocol. We present the new nonrepudiation protocol in Section 6.4, and conclude in Section 6.5.

6.1 Proof Values

Each of the proofs stored in cloud C can have two possible values. Either the proof stored is the correct proof, or it is the empty proof.

The Proof1 stored in cloud C has two possible values as follows:

Proof1 = (S,C,R,$H(F)$, signature by S) or Proof1 = empty proof

where S is the identity of party S

C is the identity of party C

R is the identity of party R

$H(F)$ is the hash of file F using a secure hash function H

the signature of S is signed using the private key of S

The Proof2 stored in cloud C has two possible values as follows:

Proof2 = (S,C,R,$H(F)$, signature by C) or Proof2 = empty proof

where S is the identity of party S

C is the identity of party C

R is the identity of party R

$H(F)$ is the hash of file F using a secure hash function H

the signature of C is signed using the private key of C

The Proof3 stored in cloud C has two possible values as follows:

Proof3 = (S,C,R,$H(F)$, signature by R) or Proof3 = empty proof

where S is the identity of party S

C is the identity of party C

R is the identity of party R

$H(F)$ is the hash of file F using a secure hash function H

the signature of R is signed using the private key of R

Next, we describe the state of the protocol

6.2 Protocol State

There are a total of three parties involved in the nonrepudiation protocol from party S to party R over a cloud C. Each of these three parties has a state associated with it. Next, we describe the state of each party.

The state of party S is the data item stored in S, which is $H(F)$. Therefore, the state of party S is as follows:

state of party S: $H(F)$

The state of party R is the data item stored in R, which is $H(F)$. Therefore, the state of party R is as follows:

state of party R: $H(F)$

The state of party C are the data items stored in C, which are $H(F)$, F, Proof1, Proof2, and Proof3. Therefore, the state of party C is as follows:

state of party C: $H(F)$, F, Proof1, Proof2, Proof3

The state of the protocol is defined as the concatenation of the states of the three parties. Therefore, the state of the protocol is defined as follows: the state of the protocol: (state of party S, state of party R, state of party C)

Note that each of the three proofs stored in cloud C either have the correct value or the empty value as described in the previous section.

In the next section, we discuss the legitimate and illegitimate states of the protocol.

6.3 Legitimate and Illegitimate States of the Protocol

The state of the protocol is called *legitimate* iff either both Proof1 and Proof3 have the value of the empty proof, or both these proofs have the value of the respective correct proof.

Next, we define the illegitimate state

The state of the protocol is *illegitimate* iff one of the two proofs, Proof1 or Proof3, has the value of the empty proof, but the other proof has the value of the correct proof.

If we recall from Chapter 5, the protocol has two actions in the second mode of the protocol. In the first action, party S asks for file F and its two needed proofs, Proof2 and Proof3. When C receives this request from S, it sends to S both file F and its two needed proofs. In the second action, party R asks for file F and its two needed proofs, Proof1 and Proof2. When C receives this request from R it sends to it both file F and its two needed proofs.

If the state of the protocol is legitimate, then each of the two parties S and R can execute the above two actions. The result of the execution of the two actions is that, party S ends up with file F and the two proofs, Proof2 and Proof3. Similarly, party R ends up with file F and the two proofs, Proof1 and Proof2. Since the state of the protocol is legitimate, then either both Proof1 and Proof3 are empty, in which case both S and R will not have the complete proof. Or both the proofs, Proof1 and Proof3 will be non-empty, in which case, depending on the value of Proof2, either both parties will end up

with needed proofs, or neither of the two parties will end up with the needed proofs. Therefore, if the state of the protocol is legitimate, then executing the first two actions, results in either both parties receiving the complete proof, or neither of these two parties receiving its complete proof.

If the state of the protocol is illegitimate, then the result of the execution of the two actions is as follows. Party S ends up with file F and the two proofs, Proof2 and Proof3. Similarly, party R ends up with file F and the two proofs, Proof1 and Proof2. Since the state of the protocol is illegitimate, which means that the value of one of the two proofs, Proof1 and Proof3, is the empty proof, while the value of the other proof is not the empty proof. Now we consider the following two cases.

Consider the case when Proof1 is empty proof, while Proof2, and Proof3 are the correct proofs. In this case, when the two actions of the protocol are executed by parties S and R, then party S will end up with file F and correct Proof2 and Proof3. On the other hand, party R will end up with file F and correct Proof2 but the value of Proof1 will be empty. In this case, S will end up with complete proof and can convince a judge that file F was indeed received by R from S. But on the other hand, R will not have its needed proofs, and will not be able to convince a judge that file F was indeed sent by S to R.

Consider the case when Proof3 is empty proof, while Proof1, and Proof2 are the correct proofs. In this case, when the two actions of the protocol are executed by parties S and R, then party R will end up with file F and correct

Proof1 and Proof2. On the other hand, party S will end up with file F and correct Proof2 but the value of Proof3 will be empty. In this case, R will end up with complete proof and can convince a judge that file F was indeed sent by S to R. But on the other hand, S will not have its needed proofs, and will not be able to convince a judge that file F was indeed received by R from S.

From the above two cases, it is clear that if the state of the protocol is illegitimate, then when the two actions of the protocol presented in chapter 5 are executed, then the above two problematic scenarios can occur. In these problematic scenarios, one of the two parties, S and R, ends up with its needed proofs, but the other party does not end up with its needed proofs violating the nonrepudiation requirements. In the next section, we present a new protocol, which is able to avoid these problematic scenarios. In the new protocol, if the state of the protocol is illegitimate, then after some finite time period, the state of the protocol changes from an illegitimate state to a legitimate state.

6.4 Stabilizing Nonrepudiation Protocol

In the previous section, we discussed that if the state of the protocol is illegitimate, then it leads to a case where the nonrepudiation requirements of the protocol are violated. In this section, we add a third action to the nonrepudiation protocol presented in chapter 5 which adds stabilization property to the protocol. The stabilization property ensures that if the state of the protocol is illegitimate, then after some finite time period, the state of the protocol changes from an illegitimate state to a legitimate state. The new

nonrepudiation protocol fulfills the two requirements as stated below:

- If the state of the protocol is legitimate, then after executing any action of the protocol, the state of the protocol remains as legitimate
- If the state of the protocol is illegitimate, then after some finite time period, the state of the protocol changes from an illegitimate state to a legitimate state.

Now we present the third action of the protocol which adds stabilization property as discussed above. We assume that cloud C has access to a clock, which times out periodically. Whenever cloud C times out, it checks whether the state of the protocol is legitimate or not. If the state of the protocol is not legitimate, then it assigns the empty proof to both the proofs, Proof1 and Proof3. The result of assigning empty proof to the two proofs, Proof1 and Proof3 is that the state of the protocol changes from an illegitimate state to a legitimate state. This is because when both Proof1 and Proof3 have the empty proof value then, both parties S and R will not be able to receive the needed proofs from C by executing the first two actions of the protocol.

Note that C can check whether the state is legitimate by checking the two proofs, Proof1 and Proof3 stored in cloud C. If either both the two proofs have the correct proof value, or both the proofs have the empty proof value, then the state of the protocol is legitimate. Otherwise the state of the protocol is illegitimate.

The third action of the protocol which is initiated by a timeout that occurs in C can be represented as follows:

if (state of protocol is not legitimate) **then**

Proof1 := empty proof

Proof3 := empty proof

end if

Next we discuss how the third action achieves the two requirements of the new nonrepudiation protocol. We consider two cases. In case one, the state of the protocol is legitimate. And in case two the state of the protocol is illegitimate.

Case 1: the state of the protocol is legitimate

If the state of the protocol is legitimate, then executing any of the three actions takes it to a legitimate state. This is becuase the first two actions result in C sending file F and the needed proofs to the two parties, S and R. These two actions do not change the state of the protocol as no change in proof values of the three proofs stored in cloud C takes place. The third action which takes place after C times out does nothing as the state of the protocol is legitimate. Therefore, the execution of the three actions of the protocol, results in the state of the protocol remaining in legitimate state.

Case 2: The state of the protocol is illegitimate:

If the state of the protocol is illegitimate, then executing any of the first two actions of the protocol will result in no change in the state of the protocol

as these two actions do not change the value of any of the proofs stored in cloud C. However, the third action which is initiated when a timeout occurs in cloud C will assign the empty proof to both the proofs, Proof1 and Proof3. The result of which is that the state of the protocol changes from an illegitimate state to a legitimate state.

From the above two cases, it is clear that the third action provides the stabilization property to the protocol.

6.5 Conclusion and Chapter Summary

In this chapter, we design a new nonrepudiation protocol from party S to party R over cloud C that has interesting stabilization property. This stabilization property ensures that as long as the state of the protocol is legitimate, then either both parties S and R receive their needed proofs or neither of the two parties receive their needed proofs. If the state of the protocol becomes illegitimate, which happens when either Proof1 or Proof3 stored in cloud C gets lost but the other data items are still stored in cloud C, then the protocol ensures that after some finite time period, the state of the protocol changes from an illegitimate state to a legitimate state.

The previous nonrepudiation protocol from S to R over cloud C which is presented in Chapter 5 leads to a compromised state when one of the two proofs, Proof1 or Proof3, stored in cloud C gets lost. In the compromised state, one party can receive its needed proofs from cloud C, but the other party is not able to receive its needed proofs from cloud C. In this chapter,

we design a new nonrepudiation protocol from S to R over C which does not suffer from this compromised state.

Chapter 7

Multiparty Nonrepudiation Protocols in Cloud Systems

The material presented in this chapter is based on the paper [1] [1]. A nonrepudiation protocol from a sender S to a set of potential receivers $\{R1, R2, \ldots, Rn\}$ performs two functions. First, this protocol enables S to send to every potential receiver Ri a copy of file F along with a proof that can convince an unbiased judge that F was indeed sent by S to Ri. Second, this protocol also enables each Ri to receive from S a copy of file F and to send back to S a proof that can convince an unbiased judge that F was indeed received by Ri from S. When a nonrepudiation protocol from S to $\{R1, R2, \ldots, Rn\}$ is implemented in a cloud system, the communications between S and the set of potential receivers $\{R1, R2, \ldots, Rn\}$ are not carried out directly. Rather, these communications are carried out through a cloud C. In this chapter, we present a nonrepudiation protocol that is implemented in a cloud system and show that this protocol is correct. We also show that this protocol has two clear advantages over nonrepudiation protocols that are not implemented in cloud systems.

[1] Muqeet Ali did ninety percent of the work presented in the paper

We present a nonrepudiation protocol for transferring a file F from a sender S to a set of potential receivers $\{R1, R2, \ldots, Rn\}$. Each potential receiver Ri can decide on its own whether or not to receive the sent file F. The protocol is designed to satisfy the following two requirements, called nonrepudiation requirements:

1. If a potential receiver Ri decides to receive file F, then Ri ends up obtaining both file F and a proof that F was indeed sent by S to Ri. In this case, S also ends up obtaining a proof that file F was indeed received by Ri from S.

2. If a potential receiver Ri decides not to receive file F, then Ri ends up not obtaining both F and a proof that F was indeed sent by S to Ri, and S ends up not obtaining a proof that F was received by Ri from S.

From Requirement 1, if a potential receiver Ri decides to receive file F, then (i) Ri ends up obtaining a proof that F was indeed sent by S to Ri and S can't repudiate correctly that it has sent F to Ri, and (ii) S ends up obtaining a proof that F was indeed received by Ri from S and Ri can't repudiate correctly that it has received F from S.

From Requirement 2, if a potential receiver Ri decides not to receive file F, then (i) Ri ends up not obtaining a proof that F was sent by S to Ri and S can repudiate correctly that it has sent F to Ri, and (ii) S ends up

not obtaining a proof that F was received by Ri from S and Ri can repudiate correctly that it has received F from S.

Our protocol does not support direct communications between sender S and the potential receivers $R1$, $R2$, ..., and Rn. Rather, the communications between the sender and the potential receivers are carried out over a cloud C. For example, a file F is sent from S to $R1$, $R2$, ..., and Rn in several steps. In the first step, sender S sends file F to cloud C. In the second step, cloud C informs the potential receivers that it can forward file F to each of them upon request. In the third step, any potential receiver Ri can decide to receive file F and asks cloud C to forward file F to Ri. In the fourth step, cloud C forwards file F to every potential receiver that has requested F.

The nonrepudiation protocol that we present in this chapter has two clear advantages over prior nonrepudiation protocols, e.g. [29, 39, 62, 65], that do not involve a cloud.

First, in our protocol, sender S and each potential receiver Ri sends at most one message during execution of the protocol, regardless of the number of potential receivers involved in the protocol. (However, as discussed below, cloud C in our protocol sends at most $3n$ messages, where n is the number potential receivers involved in the protocol.)

Second, in our protocol, each sent file F is stored only in the cloud. In prior nonrepudiation protocols that do not involve a cloud, each sent file F is stored in the sender and in each potential receiver that decided to receive F.

7.1 Protocol Specification

Our nonrepudiation protocol involves $(n+2)$ parties: a sender S, a cloud C, and n potential receivers $R1$, $R2$, ..., and Rn. Each one of these parties has a private key and a public key. Each party knows its private and public keys and knows the public keys of all other parties.

Any message that is supposed to be sent by one party and to be received by another party during the execution of the nonrepudiation protocol can be lost and not received by the other party. We assume that the nonrepudiation protocol is to be executed on top of a reliable transport service (such as TCP in the internet). This transport service takes any message that is sent by any party during the execution of the protocol and transmits this message any number of times until the message is ultimately received by its intended party. Therefore, message loss due to an unreliable communication medium is not possible.

The protocol enables sender S to send a file F over cloud C to some or all of the potential receivers $R1$, $R2$, ..., and Rn. It is up to each potential receiver Ri to decide whether or not to receive the sent file F. We assume that file F contains the identity of the sender S and the identities of all potential receivers $R1$, $R2$, ..., Rn.

The protocol consists of the following five steps:

1. Store File Step:

In this step, sender S sends a message that contains file F to cloud C. This message requests that C stores F and later forwards it to every potential receiver Ri that requests F.

2. Announce File Step:

File F which is received by cloud C in Step 1 contains the identity of each potential receiver of F. Thus, in Step 2, cloud C sends a message to every potential receiver of F announcing that each potential receiver can request to receive file F from C. If a potential receiver Ri decides that it does not need to receive file F from C, then Ri terminates without executing the remaining steps of the protocol. On the other hand, if a potential receiver Ri decides that it needs to receive file F from C, then this Ri proceeds to execute the remaining steps of the protocol.

3. Request File Step:

A potential receiver Ri which decided in Step 2 that it needs to receive file F from C sends a message to C requesting that C sends file F to Ri.

4. Send File Step:

Cloud C sends a message that contains file F to a potential receiver Ri that requested to receive file F from C in Step 3.

5. File Received Step:

Cloud C sends a message to sender S. This message informs S that (1) C received file F from S in Step 1 and (2) a potential receiver Ri which requested to receive file F from C in Step 3 later received file F from C in Step 4.

Note that if a potential receiver Ri decides that it does not need to receive file F from cloud C, then this Ri will terminate its execution at Step 2. However if Ri decides that it needs to receive file F from C, then this Ri will terminate its execution at Step 4.

For this protocol to be a nonrepudiation protocol, execution of the protocol needs to satisfy the following two requirements, called nonrepudiation requirements, for each potential receiver Ri:

NR1. If during execution of the protocol Ri decides to receive file F from cloud C, then Ri ends up obtaining both F and a proof that F was indeed sent by S to Ri. Also S ends up obtaining a proof that F was indeed received by Ri from S.

NR2. If during execution of the protocol Ri decides not to receive file F from cloud C, then Ri ends up not obtaining F and not obtaining a proof that F was sent by S to Ri. Also S ends up not obtaining a proof that F was received by Ri from S.

The two nonrepudiation requirements NR1 and NR2 mention two proofs: a proof that F was sent by S to Ri and a proof that F was received by Ri from S. However because our protocol does not support direct communications between S and Ri, it is more convenient to replace each one of these

proofs by two proofs.

Therefore, the proof that F was sent by S to Ri is replaced by the following two proofs:

Proof 1: A proof that F was sent by S to C

Proof 2: A proof that F was sent by C to Ri

Also, the proof that F was received by Ri from S is replaced by the following two proofs:

Proof 3: A proof that F was received by Ri from C

Proof 4: A proof that F was received by C from S

As discussed below, Proof 1 is computed by S and Proof 2 is computed by C, then both these proofs are ultimately sent to Ri. Later, Ri can combine these two proofs and end up with a proof that F was sent by S to Ri as dictated by the nonrepudiation requirement NR1.

Similarly, Proof 3 is computed by Ri and proof 4 is computed by C, then both these proofs are ultimately sent to S. Therefore, S can combine these two proofs and end up with a proof that F was received by Ri from S as dictated by the nonrepudiation requirement NR1.

The five steps of the protocol can now be refined to show how the four proofs are exchanged between S, C, and Ri during execution of the protocol:

1. Store File Step:

In this step, sender S sends a message that contains both file F and Proof 1

to cloud C. This message requests that C stores F and later forwards it to every potential receiver Ri that requests F.

2. Announce File Step:

File F which is received by cloud C in Step 1 contains the identity of each potential receiver of F. Thus, in Step 2, cloud C sends a message to every potential receiver of F announcing that each potential receiver can request to receive file F from C. If a potential receiver Ri decides that it does not need to receive file F from C, then Ri terminates without executing the remaining steps of the protocol. On the other hand, if a potential receiver Ri decides that it needs to receive file F from C, then Ri proceeds to execute the remaining steps of the protocol.

3. Request File Step:

A potential receiver Ri which decided in Step 2 that it needs to receive file F from C sends a message that contains Proof 3 to C requesting that C sends file F to Ri.

4. Send File Step:

Cloud C sends a message that contains both file F and Proof 2 to a potential receiver Ri that requested to receive file F from C in Step 3.

5. File Received Step:

Cloud C sends a message that contains both Proof 3 and Proof 4 to sender S. This message informs S that (1) C received file F from S in Step 1 and (2) a potential receiver Ri which requested to receive file F from C in Step 3 later received file F from C in Step 4.

The five steps of the nonrepudiation protocol can now be summarized as follows:

1. Store File Step:

$S \rightarrow C$: (F, Proof 1)

In this step, S computes Proof 1 and sends file F and Proof 1 to C.

2. Announce File Step:

$\{R1, R2, \ldots, Rn\} \leftarrow C$: Proof 1

In this step, C forwards Proof 1 to every potential receiver of F. If some Ri decides that it does not need to receive file F, then this Ri does not execute the remaining steps of the protocol. Otherwise, Ri proceeds to execute the remaining steps of the protocol.

3. Request File Step:

$Ri \rightarrow C$: Proof 3

In this step, Ri computes Proof 3 and sends it to C.

4. Send File Step:

$Ri \leftarrow C$: (F, Proof 2)

In this step, C computes Proof 2 and sends file F (received in Step 1) and Proof 2 to Ri which requested file F in Step 3.

5. File Received Step:

$S \leftarrow C$: (Proof 3, Proof 4)

In this step, C computes Proof 4 and sends both Proof 3 (which C received in Step 3) and Proof 4 to S.

7.2 Protocol Verification

In this section, we prove that any execution of the protocol specified in the previous section satisfies the two nonrepudiation requirements NR1 and NR2 for each potential receiver Ri.

Proof of NR1: Assume that during some execution of the protocol, Ri decides to receive file F from cloud C. In this case, Ri proceeds to execute Steps 2, 3, and 4, then S proceeds to execute Step 5. Thus, Ri ends up receiving Proof 1 in Step 2 and receiving file F and Proof 2 in Step 4. As mentioned earlier, Ri can combine the two Proofs 1 and 2 to construct a proof that file F was indeed sent by S to Ri. Therefore, Ri ends up obtaining both file F and a proof that file F was indeed sent by S to Ri. Similarly, S ends up receiving the two Proofs 3 and 4. As mentioned earlier, S can combine these

two proofs to construct a proof that file F was indeed received by Ri from S. Therefore, S ends up obtaining a proof that file F was indeed received by Ri from S. This execution of the protocol satisfies the nonrepudiation requirement NR1.

Proof of NR2: Assume that during some execution of the protocol, Ri decides not to receive file F from cloud C. In this case, Ri terminates immediately after executing Step 2 and never gets to execute Step 3 or Step 4. Thus, Ri never gets to receive file F and never gets to receive Proof 2 before it terminates. Because Ri needs Proof 2 (along with Proof 1) to construct a proof that F was indeed sent by S to Ri, Ri ends up not obtaining file F and not obtaining a proof that F was sent by S to Ri. Similarly, S never gets to execute Step 5 and never gets to receive Proof 3 or Proof 4. Because S needs both Proofs 3 and 4 to construct a proof that file F was indeed received by Ri from S, S ends up not obtaining a proof that F was indeed received by Ri from S. This execution of the protocol satisfies the nonrepudiation requirement NR2.

7.3 Computing the Four Proofs

In this section, we discuss how each of the four proofs (namely Proof 1, 2, 3, and 4) is computed during any execution of the nonrepudiation protocol.

Proof 1 indicates that file F was sent by S to C. This proof can be computed as the signed tuple (S, C, $H(F)$), which is signed using the private key of S, where

S is the identity of the sender of file F

C is the identity of the receiver of file F

$H(F)$ is a hashing of file F using the secure hashing function H

The tuple (S, C, $H(F)$) is signed using the private key of S

Note that only S can compute Proof 1 because only S knows its own private key. Also, C and each potential receiver of file F can check the correctness of Proof 1 because each of them knows the public key of S. Proof 1 is computed by S and is sent to C during Step 1 of the protocol. Then this proof is later forwarded by C to every potential receiver of file F during Step 2 of the protocol.

Proof 2 indicates that file F was sent by C to a potential receiver Ri that requested to receive F from C. This proof can be computed as the signed tuple (C, Ri, $H(F)$), which is signed using the private key of C, where

C is the identity of the sender of file F

Ri is the identity of the receiver of file F

$H(F)$ is a hashing of file F using the secure hashing function H

The tuple (C, Ri, $H(F)$) is signed using the private key of C

Note that only C can compute Proof 2 because only C knows its own private

key. Also, each potential receiver of file F can check the correctness of Proof 2 because each of them knows the public key of C. Proof 2 is computed by C and is sent to a potential receiver Ri during Step 4 of the protocol provided that Ri requested to receive file F from C during Step 3 of the protocol.

Proof 3 indicates that file F was received by a potential receiver Ri from C. This proof can be computed as the signed tuple (C, Ri, $H(F)$), which is signed using the private key of Ri, where

C is the identity of the sender of file F

Ri is the identity of the receiver of file F

$H(F)$ is a hashing of file F using the secure hashing function H

The tuple (C, Ri, $H(F)$) is signed using the private key of Ri

Note that only Ri can compute Proof 3 because only Ri knows its own private key. Also, each of C and S can check the correctness of Proof 3 because each of them knows the public key of each Ri. Proof 3 is computed by Ri and is sent to C during Step 3 of the protocol. Then this proof is forwarded by C to S during Step 5 of the protocol.

Proof 4 indicates that file F was received by C from S. This proof can be computed as the signed tuple (S, C, $H(F)$), which is signed using the private key of C, where

S is the identity of the sender of file F

C is the identity of the receiver of file F

$H(F)$ is a hashing of file F using the secure hashing function H

The tuple (S, C, $H(F)$) is signed using the private key of C

Note that only C can compute Proof 4 because only C knows its own private key. Also, S can check the correctness of Proof 4 because S knows the public key of C. Proof 4 is computed by C and is sent to S during Step 5 of the protocol.

7.4 Communication Complexity of the Protocol

In this section, we compute an upper bound on the number of messages that are sent (and received) during execution of the nonrepudiation protocol.

Step 1 is executed once, and one message is sent when this step is executed.

Step 2 is executed once, and n messages are sent when this step is executed.

Step 3 is executed at most n times, and one message is sent when this step is executed.

Step 4 is executed at most n times, and one message is sent when this step is executed.

Step 5 is executed at most n times, and one message is sent when this

step is executed.

It follows that at most $4n+1$ messages are sent during execution of the nonrepudiation protocol. Of these messages, only one message is sent by sender S (in Step 1), at most one message is sent by each potential receiver Ri (in Step 3), and at most $3n$ messages are sent by cloud C (in Steps 2, 4, and 5).

Now consider any nonrepudiation protocol that does not involve a cloud. This protocol involves only a sender S and n potential receivers $R1$, $R2$, ..., Rn. (Examples of such protocols are presented in [29, 39].) During execution of this protocol, sender S needs to send at least one message to each potential receiver Ri in order to ensure that Ri obtains a proof that file F has been sent by S to Ri. Also, each potential receiver Ri needs to send at least one message to sender S in order to ensure that S obtains a proof that file F has been received by Ri from S.

Therefore, one advantage of our nonrepudiation protocol that involves a cloud over any nonrepudiation protocol that does not involve a cloud is to reduce the number of messages that sender S needs to send during execution of the protocol from n messages to a single message.

7.5 The Cloud as a Permanent Storage of Files

When execution of the nonrepudiation protocol terminates, cloud C ends up with the following two items:

File F

The hash $H(F)$ of file F

We assume that cloud C keeps these two items indefinitely.

Similarly, when execution of the nonrepudiation protocol terminates, sender S ends up with four items for each potential receiver Ri that has requested to receive file F in Step 3:

File F

The hash $H(F)$ of file F

Proof 3

Proof 4

Note that the potential receiver Ri that requested to receive file F in Step 3 is named in Proof 3. Also note that S needs to keep these four items indefinitely because at any time in the future S may need to use these four items to prove to a judge that file F was indeed received by some potential receiver Ri (named in Proof 3) from S.

Fortunately, because of our assumption that cloud C keeps the two items F and $H(F)$ indefinitely, sender S does not need to keep file F indefinitely. Rather, when S needs to get file F, it gets file F from cloud C. Therefore, sender S needs to indefinitely keep the following three items for each potential receiver Ri that has requested to receive file F in Step 3:

The hash $H(F)$ of file F

Proof 3

Proof 4

Similarly, when execution of the nonrepudiation protocol terminates, each potential receiver Ri that has requested to receive file F in Step 3 ends up with the following four items:

File F

The hash $H(F)$ of file F

Proof 1

Proof 2

Note that each Ri that has requested to receive file F in Step 3 needs to keep these four items indefinitely because at any time in the future Ri may need to use these four items to prove to a judge that file F was indeed sent by S to Ri.

Fortunately, because of our assumption that cloud C keeps the two items F and $H(F)$ indefinitely, each potential receiver Ri that has requested to receive file F in Step 3 does not need to keep file F indefinitely. Rather, when Ri needs to get file F, it gets file F from cloud C. Therefore, each Ri that has requested to receive file F in Step 3 needs to indefinitely keep the following three items:

The hash $H(F)$ of file F

Proof 1

Proof 2

7.6 Conclusion and Chapter Summary

In this chapter, we presented a nonrepudiation protocol for transferring files from a sender S to a set of potential receivers $\{R1,R2, \ldots, Rn\}$. The presented protocol does not support direct communications between the sender and the potential receivers. Rather, the communications between the sender and the potential receivers are carried out through a cloud C.

For convenience, let P denote the nonrepudiation file transfer protocol presented in this chapter. Also let Q denote any nonrepudiation file transfer protocol that does not involve a cloud. Then, protocol P has two advantages over protocol Q. First, to transfer one file F (from the sender to the potential receivers), the sender in protocol P ends up sending only one message, whereas the sender in protocol Q ends up sending n messages, where n is the number of potential receivers of file F. Second, protocol P needs to store only one copy of each transferred file, whereas protocol Q needs to store $(n+1)$ copies of each transferred file. (The stored copy of each transferred file in protocol P is stored in cloud C, and the stored $(n+1)$ copies of each transferred file in protocol Q are stored in the sender and in the potential receivers of the file.)

In the nonrepudiation file transfer protocol presented in this chapter, execution of each potential receiver Ri is guaranteed to terminate but execution of sender S is not guaranteed to terminate. Specifically, execution of Ri either terminates after executing Step 2 (if Ri decides not to receive file F from the cloud C) or terminates after executing Step 4 (if Ri decides to receive file F from cloud C). However, if Ri decides not to receive file F from

cloud C, then execution of Step 5 in the protocol never terminates and sender S will continue to wait indefinitely for a "file F received message" that will never arrive. This problem can be solved by assuming that there is an upper bound of T seconds on the elapsed time after execution of Step 1 is terminated and before execution of Step 5 is terminated. Now, after sender S executes Step 1, it waits to receive a "file F received message" in Step 5. But if S waits for T seconds without receiving the "file F received message", then S concludes that the "file F received message" has not been sent and will not be sent (indicating that Ri has decided not to receive file F from the cloud C), and so S terminates.

Chapter 8

Conclusion and Future Work

In this dissertation, we have presented many novel contributions to the design of nonrepudiation protocols, which we refer to as the second generation of nonrepudiation protocols. The inspiration behind designing second generation of nonrepudiation protocols was based on several limitations and drawbacks of the first generation of these protocols. We have addressed many limitations and drawbacks in this dissertaion. In Chapter 3, we presented two-phase nonrepudiation protocols, which are easy to specify and verify as compared to a nonrepudiation protocol which is not two-phase. The two-phase protocols are inherently deterministic, and do not make use of real-time clocks. These characteristics make the specification and verification of these protocols much easier.

In Chapter 4, we relax the requirement of not making use of real-time clocks, and assume that party R is allowed to keep a real-time clock. In Chapter 4, we present the first nonrepudiation protocol from party S to party R that does not involve a trusted third party T which only makes a set of reasonable assumptions. The design of this protocol answered an important question in the design of nonrepudiation protocols which asked whether a

nonrepudiation protocol can be designed which only involves party S and party R and does not rely on a set of unreasonable assumptions.

Chapter 5 presents a nonrepudiation protocol for transferring file F from party S to party R which is implemented in a cloud C. This protocol is designed such that all communication takes place through a cloud C, and each party can receive its needed proof. In this protocol the only copy of file F and the needed proofs is stored in cloud C.

In Chapter 6, we design a nonrepudiation protocol from S to R where some of the proofs stored in cloud C get lost. We first show that the previous nonrepudiation protocol in Chapter 5 suffers from a compromised state when some of the proofs stored in cloud C get lost. We then present a new nonrepudiation protocol which does not suffer from this compromised state.

In Chapter 7, we design a nonrepudiation protocol for transferring files from a sender S to a subset of potential receivers $\{R.1, R.2, \ldots, R.n\}$ over a cloud C which is a generalization of the nonrepudiation protocol presented in Chapter 5.

Although we have made a number of contributions to the design of nonrepudiation protocols, several avenues of future work still remain. Below we discuss some of the avenues for future work:

1. In Chapter 6, we have presented a nonrepudiation protocol that has nice stabilization property. We have only presented this protocol for the case of three parties: party S, party R, and party C. One natural ques-

tion is how to extend this protocol to work for the case of $n+2$ parties: party S, party $R.1$, $R.2$, ..., $R.n$, and party C.

2. In Chapter 4, we presented a nonrepudiation protocol from S to R that does not involve a trusted third party T. The presented protocol in that Chapter only takes into consideration a single message m that needs to be sent from party S to party R. One could further investigate whether the presented protocol can be used to design a nonrepudiated negotiation protocol where there is an exchange of a sequence of messages $m.1$, $m.2$, ..., $m.n$ where each even-numbered message is sent from S to R, and each odd-numbered message is sent from R to S. Each party wants a proof of the exchange of messages. In this case, one needs to investigate whether a single execution of the presented protocol is sufficient to design such a nonrepudiated negotiation protocol or not?

3. Another area of future work is to design a certified email protocol that does not involve a trusted third party. One could try to adapt the nonrepudiation protocol presented in Chapter 4 in order to design a certified email protocol that does not involve a trusted third party. It would be interesting to see how the necessary and sufficient conditions presented in Chapter 4 change when designing a new protocol for the certified email.

Bibliography

[1] Muqeet Ali and Mohamed Gouda. Nonrepudiation protocols in cloud systems. In *Proceedings of the 7th International Conference on Computing Communication and Networking Technologies*, page 23. ACM, 2016.

[2] Muqeet Ali, Rezwana Reaz, and Mohamed Gouda. Two-phase nonrepudiation protocols. In *Proceedings of the 7th International Conference on Computing Communication and Networking Technologies*, page 22. ACM, 2016.

[3] Muqeet Ali, Rezwana Reaz, and Mohamed G Gouda. Nonrepudiation protocols without a trusted party. In *International Conference on Networked Systems*, pages 1–15. Springer, 2016.

[4] Nadarajah Asokan, Matthias Schunter, and Michael Waidner. Optimistic protocols for fair exchange. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 7–17. ACM, 1997.

[5] Nadarajah Asokan, Victor Shoup, and Michael Waidner. Asynchronous protocols for optimistic fair exchange. In *IEEE Symposium on Security and Privacy*, pages 86–99. IEEE, 1998.

[6] Giuseppe Ateniese, Breno de Medeiros, and Michael T Goodrich. Tricert: A distributed certified e-mail scheme. In *NDSS*, volume 1, pages 47–58.

Citeseer, 2001.

[7] Birgit Baum-Waidner. Optimistic asynchronous multi-party contract signing with reduced number of rounds. In *International Colloquium on Automata, Languages, and Programming*, pages 898–911. Springer, 2001.

[8] Birgit Baum-Waidner and Michael Waidner. Round-optimal and abuse-free optimistic multi-party contract signing. In *International Colloquium on Automata, Languages, and Programming*, pages 524–535. Springer, 2000.

[9] Chakib Bekara, Thomas Luckenbach, and Kheira Bekara. A privacy preserving and secure authentication protocol for the advanced metering infrastructure with non-repudiation service. *Proc. of ENERGY*, 2012.

[10] Giampaolo Bella and Lawrence C Paulson. Mechanical proofs about a non-repudiation protocol. In *International Conference on Theorem Proving in Higher Order Logics*, pages 91–104. Springer, 2001.

[11] Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, 1990.

[12] Dan Boneh and Moni Naor. Timed commitments. In *Annual International Cryptology Conference*, pages 236–254. Springer, 2000.

[13] Jan Cederquist, Ricardo Corin, and M Torabi Dashti. On the quest for impartiality: Design and analysis of a fair non-repudiation protocol. In

International Conference on Information and Communications Security, pages 27–39. Springer, 2005.

[14] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369. ACM, 1986.

[15] Mohammad Torabi Dashti, Jan Cederquist, and Yanjing Wang. Risk balance in optimistic non-repudiation protocols. In *International Workshop on Formal Aspects in Security and Trust*, pages 263–277. Springer, 2011.

[16] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[17] Jun Feng and Yu Chen. A fair non-repudiation framework for data integrity in cloud storage services. *International Journal of Cloud Computing*, 2(1):20–47, 2013.

[18] Jun Feng, Yu Chen, Wei-Shinn Ku, and Pu Liu. Analysis of integrity vulnerabilities and a non-repudiation protocol for cloud data storage platforms. In *Parallel Processing Workshops (ICPPW)*, pages 251–258. IEEE, 2010.

[19] Jun Feng, Yu Chen, Douglas Summerville, Wei-Shinn Ku, and Zhou Su. Enhancing cloud storage security against roll-back attacks with a new fair

multi-party non-repudiation protocol. In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 521–522. IEEE, 2011.

[20] Josep Lluís Ferrer-Gomila, Magdalena Payeras-Capellà, and Llorenç Huguet-Rotger. A realistic protocol for multi-party certified electronic mail. In *International Conference on Information Security*, pages 210–219. Springer, 2002.

[21] Josep Lluís Ferrer-Gomila, Magdalena Payeras-Capellà, and Llorenç Huguet i Rotger. An efficient protocol for certified electronic mail. In *International Workshop on Information Security*, pages 237–248. Springer, 2000.

[22] Juan A Garay, Markus Jakobsson, and Philip MacKenzie. Abuse-free optimistic contract signing. In *Annual International Cryptology Conference*, pages 449–466. Springer, 1999.

[23] Juan A Garay and Philip MacKenzie. Abuse-free multi-party contract signing. In *International Symposium on Distributed Computing*, pages 151–166. Springer, 1999.

[24] Sigrid Gürgens and Carsten Rudolph. Security analysis of (un-) fair non-repudiation protocols. In *Formal aspects of security*, pages 97–114. Springer, 2003.

[25] Sigrid Gürgens, Carsten Rudolph, and Holger Vogt. On the security of fair non-repudiation protocols. In *International Conference on Information Security*, pages 193–207. Springer, 2003.

[26] Jorge L Hernandez-Ardieta, Ana I Gonzalez-Tablas, and Benjamin Ramos Alvarez. An optimistic fair exchange protocol based on signature policies. *computers & security*, 27(7):309–322, 2008.

[27] Gwan-Hwan Hwang, Jenn-Zjone Peng, and Wei-Sian Huang. A mutual nonrepudiation protocol for cloud storage with interchangeable accesses of a single account from multiple devices. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 439–446. IEEE, 2013.

[28] Kwangjo Kim, Sangjoon Park, and Joonsang Baek. Improving fairness and privacy of zhou-gollmann's fair non-repudiation protocol. In *International Workshops on Parallel Processing*, pages 140–145. IEEE, 1999.

[29] Steve Kremer and Olivier Markowitch. A multi-party non-repudiation protocol. In *Information Security for Global Information Infrastructures*, pages 271–280. Springer, 2000.

[30] Steve Kremer and Olivier Markowitch. Fair multi-party non-repudiation protocols. *International Journal of Information Security*, 1(4):223–235, 2003.

[31] Steve Kremer, Olivier Markowitch, and Jianying Zhou. An intensive survey of fair non-repudiation protocols. *Computer communications*, 25(17):1606–1621, 2002.

[32] Steve Kremer and Jean-François Raskin. A game-based verification of non-repudiation and fair exchange protocols. In *International Conference on Concurrency Theory*, pages 551–565. Springer, 2001.

[33] Maria Krotsiani and George Spanoudakis. Continuous certification of non-repudiation in cloud storage services. In *IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 921–928. IEEE, 2014.

[34] Panagiota Lagou and Gregory P Chondrokoukis. Biometrics for non-repudiation: security and legal aspect. *International Journal of Biometrics*, 3(2):128–147, 2011.

[35] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina. Automatic analysis of a non-repudiation protocol. *Electronic Notes in Theoretical Computer Science*, 112:113–129, 2005.

[36] Nam-Yih Lee, Chi-Chao Chang, Chun-Li Lin, and Tzonelih Hwang. Privacy and non-repudiation on pay-tv systems. *IEEE Transactions on Consumer Electronics*, 46(1):20–27, 2000.

[37] Guoqiang Li and Mizuhito Ogawa. On-the-fly model checking of fair non-repudiation protocols. In *International Symposium on Automated Technology for Verification and Analysis*, pages 511–522. Springer, 2007.

[38] Shiqun Li, Guilin Wang, Jianying Zhou, and Kefei Chen. Fair and secure mobile billing systems. *Wireless personal communications*, 51(1):81–93,

2009.

[39] Olivier Markowitch and Steve Kremer. A multi-party optimistic non-repudiation protocol. In *International Conference on Information Security and Cryptology*, pages 109–122. Springer, 2000.

[40] Olivier Markowitch and Steve Kremer. An optimistic non-repudiation protocol with transparent trusted third party. In *International Conference on Information Security*, pages 363–378. Springer, 2001.

[41] Olivier Markowitch and Yves Roggeman. Probabilistic non-repudiation without trusted third party. In *Second Conference on Security in Communication Networks*, volume 99, pages 25–36. Amalfi, Italy: IEEE Communication Society, 1999.

[42] Aybek Mukhamedov and Mark Ryan. Improved multi-party contract signing. In *International Conference on Financial Cryptography and Data Security*, pages 179–191. Springer, 2007.

[43] Jose A Onieva, Javier Lopez, Rodrigo Roman, Jianying Zhou, and Stefanos Gritzalis. Integration of non-repudiation services in mobile drm scenarios. *Telecommunication Systems*, 35(3-4):161–176, 2007.

[44] Jose A Onieva, Jianying Zhou, and Javier Lopez. Multiparty nonrepudiation: A survey. *ACM Computing Surveys (CSUR)*, 41(1):5, 2009.

[45] Jose Antonio Onieva, Jianying Zhou, Mildrey Carbonell, and Javier Lopez. A multi-party non-repudiation protocol for exchange of different messages.

In *Security and Privacy in the Age of Uncertainty*, pages 37–48. Springer, 2003.

[46] Rolf Oppliger. Certified mail: the next challenge for secure messaging. *Communications of the ACM*, 47(8):75–79, 2004.

[47] Rolf Oppliger. Providing certified mail services on the internet. *IEEE Security & Privacy*, 5(1), 2007.

[48] Chung-Ming Ou and Chung-Ren Ou. Adaptation of proxy certificates to non-repudiation protocol of agent-based mobile payment systems. *Applied Intelligence*, 30(3):233–243, 2009.

[49] Chung-Ming Ou and Chung-Ren Ou. Adaptation of agent-based non-repudiation protocol to mobile digital right management (drm). *Expert Systems with Applications*, 38(9):11048–11054, 2011.

[50] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Optimal efficiency of optimistic contract signing. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 113–122. ACM, 1998.

[51] Raluca Ada Popa, Jacob R Lorch, David Molnar, Helen J Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *USENIX Annual Technical Conference*, volume 242, page 14, 2011.

[52] Michael O Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.

[53] Judson Santiago and Laurent Vigneron. Optimistic non-repudiation protocol analysis. In *IFIP International Workshop on Information Security Theory and Practices*, pages 90–101. Springer, 2007.

[54] Bruce Schneier and James Riordan. A certified e-mail protocol. In *Computer Security Applications Conference*, pages 347–352. IEEE, 1998.

[55] Ronggong Song and Larry Korba. Pay-tv system with strong privacy and non-repudiation protection. *IEEE Transactions on Consumer Electronics*, 49(2):408–413, 2003.

[56] Ronggong Song and Michael R Lyu. Analysis of privacy and non-repudiation on pay-tv systems. *IEEE Transactions on Consumer Electronics*, 47(4):729–733, 2001.

[57] Jinyuan Sun, Chi Zhang, and Yuguang Fang. An id-based framework achieving privacy and non-repudiation in vehicular ad hoc networks. In *Military Communications Conference*, pages 1–7. IEEE, 2007.

[58] Kun Wei and James Heather. A theorem-proving approach to verification of fair non-repudiation protocols. In *International Workshop on Formal Aspects in Security and Trust*, pages 202–219. Springer, 2006.

[59] Wei Wu, Jianying Zhou, Yang Xiang, and Li Xu. How to achieve non-repudiation of origin with privacy protection in cloud computing. *Journal of Computer and System Sciences*, 79(8):1200–1213, 2013.

[60] Zhifeng Xiao, Yang Xiao, and DH-C Du. Non-repudiation in neighborhood area networks for smart grid. *IEEE Communications Magazine*, 51(1):18–26, 2013.

[61] Jianying Zhou. On the security of a multi-party certified email protocol. In *International Conference on Information and Communications Security*, pages 40–52. Springer, 2004.

[62] Jianying Zhou and Dieter Gollman. A fair non-repudiation protocol. In *IEEE Symposium on Security and Privacy*, pages 55–61. IEEE, 1996.

[63] Jianying Zhou and Dieter Gollmann. Certified electronic mail. In *European Symposium on Research in Computer Security*, pages 160–171. Springer, 1996.

[64] Jianying Zhou and Dieter Gollmann. Observations on non-repudiation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 133–144. Springer, 1996.

[65] Jianying Zhou and Dieter Gollmann. An efficient non-repudiation protocol. In *Computer Security Foundations Workshop*, pages 126–132. IEEE, 1997.

[66] Jianying Zhou and Kwok-Yan Lam. Undeniable billing in mobile communication. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 284–290. ACM, 1998.

[67] Jianying Zhou, Jose Onieva, and Javier Lopez. Optimized multi-party certified email protocols. *Information management & computer security*, 13(5):350–366, 2005.

[68] Jianying Zhou, Jose A Onieva, and Javier Lopez. A synchronous multi-party contract signing protocol improving lower bound of steps. In *IFIP International Information Security Conference*, pages 221–232. Springer, 2006.

Vita

Muqeet Ali was born in Nowshera, Pakistan in 1986. He completed his high school education from Rawalpindi, Pakistan. In 2004, he enrolled at Lahore University of Management Sciences (LUMS) in Lahore, Pakistan to study Computer Science and Mathematics. He graduated in May 2009 and moved to USA to pursue graduate studies at University of Texas at Austin in August 2010. He graduated with a Masters degree in Computer Science from UT Austin in December 2013. He has done research in security protocols under the supervision of Prof. Mohamed Gouda, and has research interests in the areas of networking, security protocols, and distributed computing.

Email address: *muqeet@utexas.edu*

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.